UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

LEVEL II

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 79-162T	2. GOVT ACCESSION NO. AD-A091528	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Design and Emulation of a System Kernel for X-Tree		5. TYPE OF REPORT & PERIOD COVERED Thesis
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) R. Neil Wasoner		8. CONTRACT OR GRANT NUMBER(s) (12) 126
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT Student at: UC Berkeley		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433		12. REPORT DATE (11) 30 March 79
		13. NUMBER OF PAGES 50
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) (14) AFIT-CI-79-162T		15. SECURITY CLASS. (of this report) UNCLASS
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

APPROVED FOR PUBLIC RELEASE AFR 190-17.

Fredric C. Lynch

FREDRIC C. LYNCH, Major, USAF

Director of Public Affairs

23 SEP 1980

18. SUPPLEMENTARY NOTES

Approved for public release: IAW AFR 190-17

Air Force Institute of Technology (ATC)

Wright-Patterson AFB, OH 45433

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

DTIC ELECTE NOV 10 1980 S E

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Attached

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

012200 80 10 ...

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

THE DESIGN AND EMULATION OF A SYSTEM KERNEL FOR X-TREE

R Neil Wagoner

UC Berkeley

ABSTRACT

X-tree is tree-structured network of single-chip processors designed for 1985 technology. An operating system kernel for supervising the operation of each processor was designed and coded. To test the code, an X-tree emulator was written to run on the PDP 11/70 UNIX system. The kernel internal structure and the emulation techniques used are presented in this paper. The feasibility of the kernel design is demonstrated by a successful emulation of the X-tree executing a simple user program. Suggestions are also made for future extensions of this work to add more functions to the kernel and to improve the range and efficiency of the emulation.

March 30, 1979

79-162T

THE DESIGN AND EMULATION OF A SYSTEM KERNEL FOR X-TREE

R Neil Wagoner

UC Berkeley

ABSTRACT

X-tree is tree-structured network of single-chip processors designed for 1985 technology. An operating system kernel for supervising the operation of each processor was designed and coded. To test the code, an X-tree emulator was written to run on the PDP 11/70 UNIX system. The kernel internal structure and the emulation techniques used are presented in this paper. The feasibility of the kernel design is demonstrated by a successful emulation of the X-tree executing a simple user program. Suggestions are also made for future extensions of this work to add more functions to the kernel and to improve the range and efficiency of the emulation.

March 30, 1979

TABLE OF CONTENTS

March 30, 1979

1. INTRODUCTION

If the current trends in VLSI architecture continue, it will be possible to put a complete computer, including memory and communications hardware, on a single chip by the mid-1980s [6]. To explore various issues in the construction of large scale systems from these single-chip computers, such a system is currently being designed at the University of California at Berkeley. Because of the topology of the system, it was named X-tree.

A basic description of X-tree and its concepts is provided in section 2 of this paper. This account is necessarily very brief and omits details not needed for the understanding of this particular paper. A more complete treatment can be found in [2,3,8].

The goal of the work reported herein was to define and implement an operating system kernel [10] for X-tree. At the time this project was started, almost all the work done on X-tree had been on the design of the hardware architecture and the communications network. Very little work on software had been done. In fact, this has been the first real attempt to produce running software for the system.

Section 3 of this paper describes the definition and internal structure of the node kernel. In section 4, the various techniques used to test and run the kernel on the UNIX system are discussed. Finally some reflections and closing comments are presented in section 5.

2. INTRODUCTION TO X-TREE

2.1. Basic Description

X-tree is a connected network of single-chip computers. Results of various simulation studies [3] have led to the selection of a topology which is a binary tree with some additional interconnections at each level of the tree. Many different schemes for these additional interconnections have been proposed. No final selection has yet been made. Pictured below is an X-tree with 'full ring' intralevel connections.

```
                          ------
                         | 1=1b |
                          ------
                           |  |
           ----------------    ----------------
           |                                  |
     ------| - - - - - - - - - - - - - - - - -|------
     |   ------                          ------     |
     --| 2=10b |------------------------| 3=11b |--
         ------                          ------
          |    |                          |    |
      ----      ----                  ----      ----
      |            |                  |            |
 --  --| - - - - - -| - - - - - - - - -| - - - - -|------
 |  ------        ------         ------        ------   |
 --|4=100b|------|5=101b|-------|6=110b|------|7=111b|--
    ------        ------         ------        ------
    |    |        |    |         |    |        |    |
```

Each node in the tree is a full scale computer with on-chip memory and communication hardware. All external devices (disks, terminals, etc) for the system are connected to the downward links at the 'leaves' of the tree. There are no external devices directly connected to the upper nodes, except a bootstrap device connected to the root node.

The nodes in the tree are numbered using the simple scheme illustrated in the diagram. The relationship between the numbers of connected nodes has a convenient property (in the binary representation):

(1) The number of a left child is the number of the parent with an additional 0 at the least significant end.

(2) The number of a right child is the number of the parent with an additional 1 at the least significant end.

This makes any particular node easy to locate.

2.2. Inter-node Communications

In such a system, the transfer of data between nodes could be very complex and time-consuming. For example, data from node 1 to node 4 must pass though node 2. If the CPU in node 2 had to actively transfer such data through its node, much processing power would be consumed on this simple task.

To reduce this load, dat is divided into messages. At the beginning of each message is a field which contains the number of the node to which the message is being sent. Each node contains a communications controller which can route messages through the tree without direct CPU involvement.

```
                           to parent
                               |
          ----------------------------------------
         |  --------           |                 |
         | |        |      ----------            |
         | | C P U  |     |          |           |
         | |        |<--->|   Comm   |           |
         |  --------      |  Control |           |
aux  ----|----------------|          |-----|---- aux
link     |                 ----------            link
         |                  |        |           |
         |      ------------         |           |
         |      |                    |           |
          ----------------------------------------
                |                    |
             to left             to right
              child                child
```

In this way, the CPU in each node is involved only with messages for which it is the source or final destination.

Further simulations of the message traffic [9] have revealed that it will be necessary to send multiple messages

concurrently over each single link. This is accomplished by time-multiplexing the link among several logical channels, known in X-tree as "slots". The implementation of the communication controller is discussed fully in [8]. For the purposes of this paper, it is sufficient to note that each slot is seen by the CPU as a separate device, independent from the other slots. The interface to these slots is described in section 3.

2.3. Addressing Structure

The concept of memory in X-tree is slightly different from that seen in more conventional systems. Two forms of memory are defined:

(1) Local memory is that which exists inside each node. This memory is private to the node and cannot be directly accessed by any other CPU. It is in the form of a cache and is not directly addressable.

(2) A global address space is defined over the memory devices which are attached to the leaves of the tree.

In a manner similar to virtual memory systems, local node memory is treated as pages which at any given time may contain copies of pages from the global address space.

How then are global addresses represented? The memory on the external devices is viewed as a series of "objects". Objects may be of different lengths and in many ways resemble files on conventional systems. The management of the objects on a memory device, including the mapping of objects to specific locations, is performed by the leaf node to which the device is attached.

An object in global memory is referred to by its NAME. The NAME has two parts:

(1) The node number of the leaf at which it resides, and

(2) An identifier (ID) which is unique within that leaf node, generated by the leaf node when the object is created.

A full global address is then specified by an object NAME and an offset within the object.

NAME = NODE + ID

ADDRESS = NAME + OFFSET = NODE + ID + OFFSET

In this way, no global table is needed to locate any global address. A request for data is routed to the leaf node specified in the object NAME. That leaf node then maps the ID and offset into a location on the memory device and retrieves the required page.

3. Description of the Node Kernel

3.1. Functional Description

The primary goal of the node kernel is to make possible the construction of the operating system and user programs as sets of communicating concurrent processes [1]. In particular, these processes must be able to communicate with each other using only process names, not locations within the tree.

The services provided by the kernel are necessarily quite primitive. They are concerned only with the creation and execution of processes and with the sending of messages from one process to another. The operating system layer constructed over the kernel will provide more general services for the user.

The relationship between the kernel and the upper level operating system can also be described in terms of resources managed:

1. The kernel manages node CPU time and inter-node communications.

2. The general operating system manages physical devices (disks, etc) and processor assignments.

To accomplish this, the kernel is implemented as a small set of code which exists in each node of the tree. It must perform three specific tasks:

(1) Manage the execution of processes running on that node,

(2) Manage the interface to the communications controller, and

(3) Identify messages to send them to their proper destinations.

In the discussions which follow, no distinction is made between processes which are part of the operating system and processes which perform applications. For the purposes of the kernel, all such processes are considered as user processes.

3.2. Structural Overview

As pointed out above, the kernel is a set of software which resides in each node and supervises the user processes and communication for that node. It is composed of several parts:

1. Independent kernel processes,
2. A set of device drivers for the communications channel,
3. A set of kernel service routines invoked by users processes, and
4. Various subroutines for the performance of specific tasks.

These processes and routines can be grouped into three blocks corresponding to the main tasks of the kernel:

1. Process Management: subroutines to create and start user processes; kernel service routines to be invoked by user processes.
2. Communications Management: a set of device drivers, one for each 'slot', which move messages in and out of the node.
3. Message Management: the main process of the kernel and various subroutines for manipulating messages; matches user messages to their destinations and interprets kernel messages to perform remote requests.

While these are convenient groups for understanding the structure of the kernel, it is important to realize that they are not independent of each other. For example, when kernel service routines are called by the user, they accomplish their job by in turn calling routines which are part of the message management group.

Communication between the three groups is accomplished in primarily two ways:

1. A process or routine in one group may call a routine in another group. Of course, data may be passed both ways as arguments and return values.

2. Certain processes obtain their work from FIFO queues of "work packets". Other processes place packets in these queues for later processing.

The processes of the kernel (and of the users as well) are to be executed by a hardware monitor suggested by Tim McCreary [5]. Their implementation in the emulator is described in chapter 4.

The kernel implemented for this project has ignored several sources of complexity in the X-tree. First, it is envisioned that the X-tree system is to be indefinitely extensible; hence node numbers may have an unbounded length. In order to focus on other issues, a fixed length node number of 16 bits was postulated for the kernel. This is not really a severe restriction since this length will

support a tree of 16 levels, a total of 65,535 nodes!

The second simplification is far more serious: the paging of X-tree is not emulated. Instead, all code and data structures for user processes are included directly in the emulated node that the process is running in. The addition of paging to the kernel at a later date is not absolutely straightforward, but an attempt was made to keep the code compatible with paging and the use of full global addresses. A model of user processes under the fully paged system is described in appendix E.

It is also assumed that all messages in X-tree are reliably transmitted. Later versions of this kernel will need to handle message communication failures.

3.3. Process Management

The Process Manager consists of all the code necessary to support user processes. In particular, the following parts are included:

1- Two routines for starting and creating user processes.

2- A set of kernel service routines called by user programs.

Each user process is represented by a process control block (PCB) of the form

```
 --------------------------
| link field               |-->used for the FREEPCB
|--------------------------|        linked list
| full process name        |-->identifies the process globally
|--------------------------|
| priority                 |
|--------------------------|
| status word (psw)        |-->internal process status, such as
|--------------------------|    condition codes, etc
| external status word     |-->state of the process: IN-USE,
|--------------------------|    WAITING-FOR-IO, etc
| semaphore pointer        |-->used for emulator synchronization
|--------------------------|
| IOB list pointer         |-->pointer to the process's list of
|--------------------------|    active IOBs
| name of parent           |
|--------------------------|
| program counter          |
|--------------------------|
| stack pointer            |
 --------------------------
```

A fixed number of these blocks exist in each node. This sets an effective upper limit on the number of processes active in node at one time.

Each process can be identified in two ways:

(1) By a full 32-bit name which is composed of

(a) the node number of the node that the process's work space is on, and

(b) an ID number guaranteed to be unique within that node.

(2) By a pointer (PID) to its PCB; this is used only within the local node kernel.

Translation from the local name to the full name is simple; it can be retrieved from the second field in the PCB. The other direction is a little more complicated. A hash table is maintained to hold pointers to all active PCBs in the node.

For the emulation, an additional set of kernel subprocesses is defined: one for each PCB in the node. Each can be thought of as a subprocess reserved to execute a user. At system start time, each of these subprocesses goes immediately to sleep. Also, all of the PCBs are linked together in a list of unused PCBs (the FREEPCB queue).

The creation of a new user process is performed by the subroutine KCRTUSR. Its implementation is straightforward. An unused PCB is obtained from the FREEPCB queue and is filled in with the appropriate information, specifically the priority, starting program counter, and the name of the parent process. A unique full name for the new process is

then generated, stored in the PCB and the hash_ table, and passed back to the parent process for later use.

The subroutine KSTRTUSR starts a specified user process and is equally simple. Using the hash table, the process PCB is located and the corresponding kernel subprocess (the one dedicated to that PCB) is awakened. This subprocess then

(a) Runs the user to completion,

(b) Removes the process name from the hash table,

(c) Puts the PCB back on the FREEPCB queue for later reuse, and

(d) Goes back to sleep.

Kernel Service Routines - Five functions are provided in the present kernel as service calls for the user. These are described below.

PCREATE (NODE, START, PRIO, PNAME)

Definition: A child process is created. The arguments NODE, START, and PRIO specify the node number, starting global address and priority of the new process. Once the new process has been created, its full process name will be returned to the caller in PNAME.

Implementation: If NODE is the current node number, routine KCRTUSR can be called directly. If not, a message is formed and sent to the kernel process on the proper node. The return message, which contains the new process name, is then waited for.

PSTART (PNAME)

Definition: Starts the execution of process PNAME, where PNAME is a full process name.

Implementation: Similar to PCREATE except of course that KSTRTUSR is called instead of KCRTUSR.

NPARENT (PNAME)
  Definition: Returns in PNAME the full process name of the parent of the calling process.
  Implementation: Copies it from the proper field in the calling user's PCB.

MSEND (PNAME, IONBR, LNG, MSGADDR)
  Definition: Sends a message to process PNAME; the message is taken from a buffer of length LNG starting at address MSGADDR. The IONBR is an integer which matches the IONBR in the MRECV request by the target process.
  Implementation: See section 3.4 (Message Manager).

MRECV (IONBR, LNG, MSGADDR)
  Definition: Sets up to receive a message of maximum length LNG starting at address MSGADDR. The calling process waits until the message is received. Upon return, the LNG field will contain the actual length of the message received.
  Implementation: See section 3.4 (Message Manager).

3.4. Message Management

The message manager is the heart of the kernel. It is comprised of the main kernel process and a set of subroutines for manipulating messages.

Each active message in the node is represented and managed using an I/O Control Block (IOB) with the following format.

```
 -------------------------
| queue link pointer      |-->used to link IOB into queues
|-------------------------|
| owner IOB list link     |-->link for a process's list of
|-------------------------|     active IOBs
| owner's local name      |
|-------------------------|
| I/O control nbr         |-->used to match incoming messages
|-------------------------|     with the proper MRECV request
| I/O status              |-->status of IO: IN vs OUT,
|-------------------------|     and WAITING, BUSY,or COMPLETE
| action code             |-->encodes what msg manager should
|-------------------------|     do when I/O completes
| ptr to dynamic msg bfr  |-->= NULL if no dynamic buffer used
|-------------------------|
| msg body address        |
|-------------------------|
| message length          |
|-------------------------|
| header for message      |-->see communication manager
 -------------------------
```

This block contains all the information needed to keep track of and transfer a message.

The use of the IOB and the operation of the message manager is best shown by tracing the path of a message from one user process to another.

```
                 ---------------
                 |     main      |          -------
                 |  msg mngmt    |<--|    <----------
                 |   process     |          -------       |
                 ---------------           ACTIONQ        |
                        |                                 |
                        |                                 |
                        | wake                            |
                           up                             |
        -----------     |                                 |
        |   user    |< - -                                |
        |  kernel   |                       --------      |
        |  service  |           -------    | output |     |
        |   call    |----------->      |-->|  slot  |------
        -----------             -------    | driver |
                                IOOUTQ      --------
```

The above diagram illustrates the control flow for the message in the sending node. Step-by-step:

1 - The kernel call routine, acting on behalf of the user, uses calls to message management subroutines to obtain and fill in a new IOB.

2 - Once the IOB is filled in, it is placed on the IOOUTQ, a FIFO queue of output messages waiting to be transmitted. The kernel routine then puts the user to sleep.

3 - The message is picked up by a device driver (see section 3.5, communications management) and transmitted. The IOB is then placed in ACTIONQ, the main kernel process's queue of work to be done.

4 - The main kernel process retrieves the IOB from the ACTIONQ and, guided by the ACTION CODE filled in by step 1, wakes up the user.

5 - The user process wakes up, still executing in the kernel service routine. This routine retrieves any status it needs from the IOB, then releases the IOB and returns to the user.

A slightly different sequence occurs in the receiving node:

```
                                   ---------------
                                  :     main      :          -------
                                  :  msg mngmt    :<--:    <---------
                    - - - - - - -:    process     :          -------    :
                                   ---------------           ACTIONQ    :
                   :                                                    :
                   :                                                    :
                   : wake                                               :
                     up                                                 :
   -----------     :                                                    :
  :   user    :< - -                                                    :
  :  kernel   :                                          --------       :
  :  service  :           message ------------->  : input  :            :
  :   call    :                                   :  slot  :------------
   -----------                                  ->: driver :
  : PCB    ---:-                                 :  --------
   -----------     :   -----
                   ->: IOB : - - - - - - - - - - -
                   - :     :     find match
                   :   -----
                   :
                   :   -----
                   ->: IOB :
                     :     :
                       -----
```

Step-by-step:

1 - The kernel call routine, acting on behalf of the user, uses calls to message management subroutines to obtain and fill in a new IOB.

2 - No further action is taken directly with the IOB. It is left 'hanging' in the user's list of active IOBs. The kernel routine puts the user to sleep.

3 - When a message comes into the node, it has a header which contains the target process name and an IO number (see section 3.5, communications management). The routine FINDIOB uses the hash table to find the target process's PCB and then the IOB with the matching IO number. After the message transmission is finished, the IOB is placed in ACTIONQ, the main kernel process's queue of work to be done.

4 - The main kernel process retrieves the IOB from the ACTIONQ and, guided by the ACTION CODE filled in by step 1, wakes up the user.

5 - The user process wakes up, still executing in the kernel service routine. This routine retrieves any status it needs from the IOB, then releases the IOB and returns to the user.

In addition to looking at completed IOB's to see what completion actions must be taken, the main kernel process performs another function. It is the target process for any remote service requests from other nodes. When a message containing such a request is received, the actions needed to satisfy the request are done. A reply message which contains return information is then prepared and sent.

Two remote requests are provided: create new process and start process. The formats of the message bodies are

REMOTE CREATE

```
         ------------------------------------------------
        | request | process  | process | IO nbr to |
  <---- |  code   | priority | start   | reply to  |
        |   = 1   |          | address |           |
         ------------------------------------------------
```

```
REPLY TO REMOTE CREATE
          ------------------------
          ¦ request ¦ full name ¦
     <---¦   code  ¦ of proc   ¦
          ¦   = -1  ¦ created   ¦
          ------------------------
```

```
REMOTE START
          -------------------------------------
          ¦ request ¦ full       ¦ IC nbr to ¦
     <---¦   code  ¦ process    ¦ reply to  ¦
          ¦   = 2   ¦ name       ¦           ¦
          -------------------------------------
```

```
REPLY TO REMOTE START
          ----------
          ¦ request ¦
     <---¦   code  ¦
          ¦   = -2  ¦
          ----------
```

Of course use is also made of information in the message header, so that data is not duplicated in the message body.

The resulting (simplified) cycle for the main kernel process is

```
Do forever
    Get IOB from action queue
    If it's a remote request
        Do the requested action
        Reuse the IOB to send reply
    else
        Do completion actions encoded in IOB
         (awaken user, release IOB, etc)
```

The routines to obtain and release IOBs and dynamic message buffers are also considered part of the message manager. These routines are straightforward and will not be discussed here.

3.5. Communications Management

As pointed out in section 2.2, the controller appears to the CPU as a number of independent devices called "slots". Each slot is controlled through a section of memory defined as

command byte	status byte	current transfer address	current byte count	interrupt address

Currently, the two bits READY and TERMINATE are used in the command byte. Similarly, the ERROR, END-OF-BUFFER, and END-OF-MESSAGE bits are defined in the status byte.

There are two sets of slots: one set for input and one set for output. These slots operate in a "direct memory access" fashion. When the READY command bit is on, the slot transfers data into or out of the buffer defined by the TRANSFER ADDRESS field. For each byte transferred, the TRANSFER ADDRESS is incremented and the BYTE COUNT is decremented.

Several conditions cause the transfer operation to stop:

1. An error is encountered.

2. The BYTE COUNT reaches zero (END-OF-BUFFER).

3. The end of message is found (input only).

When one of these occurs, the following steps are performed by the controller.

1. The command byte is cleared.

2. The appropriate status byte is turned on.

3. An interrupt is generated.

4. The slot waits until a new command bit is turned on.

Since a single message may be composed of multiple buffers in local memory, it will need to be transmitted in several parts. Consequently, the communications channel must be told explicitly where one message ends and the next begins. This is the function of the TERMINATE command on an output slot; it causes the termination of the current message. The next transfer operation is then interpreted as the beginning of a new message. On an input slot, the TERMINATE command causes the remainder of the current message to be flushed.

Any message has the standard format

```
       -----------------------------------------------------------
<----| message |                 message                        |
     | header  |                   body                         |
       -----------------------------------------------------------
```

and a message header looks like

```
       --------------------------------------------------------
     | target | target  |   IO   | source | source  |
<----|  node  | process |   id   |  node  | process |
     | number |  name   | number | number |  name   |
       --------------------------------------------------------
```

Of course, the "target node number" field is used by the communication hardware to route the message.

The slots are controlled by a set of drivers (implemented as processes), one for each slot. The output drivers are identical, sharing the same code and having separate local data stacks. The same is true of the input drivers.

The output drivers take their work off a common FIFO queue of messages (more strictly, IOBs) waiting to be sent out. The header and body for a message are transmitted as separate buffers. So the logic for an output drive is:

```
Do forever
     Get a message from the output queue
     Set up slot to output header
     Issue READY command and wait for interrupt
     Set up slot to output message body
     Issue READY command and wait for interrupt
     Issue TERMINATE command and wait for interrupt
     Send IOB to message manager
```

The logic for an input driver is a little more complicated. The message header must be read in before the message can be identified and the proper input buffer can be selected. Consequently, a dedicated header area is reserved fo each input slot. After the header is read in, the correct IOB to receive the message must be identified (see section 3.4 on the message manager). Once this is done, the rest is straightforward:

```
Do forever
    set up slot to read into header area
     and issue READY command
    wait for interrupt
    locate matching IOB
        (or create one if target is kernel)
    copy header into IOB
    set up slot to read into message buffer
     and issue READY command
    wait for interrupt
    send IOB to message manager
```

After both input and output operations, the message IOB is sent back to the message handler for any closing actions that may be required.

4. THE EMULATION PROGRAM

4.1. X-Tree Nodes as UNIX Processes

In order to test and debug the node kernel, some method of running it on existing hardware is needed. At this university, the most convenient way to do this is to run an of emulation on the UNIX system. However, the emulation cannot be done as a simple UNIX program. Two levels of parallel processing are needed:

(1) The X-tree has more than one node. A way is needed to model the simultaneous execution of the X-nodes and the communication between them.

(2) The kernel of the X-node, together with the user processes running on that node, represent a set of concurrent subprocesses executing within the node.

The two types of parallelism are different in one very important respect: interprocess communication. The X-nodes have no common local memory through which to communicate. The kernel subprocesses, on the other hand, rely heavily on memory structures that are in the local node and are available to all subprocesses at that node.

This observation leads to a relatively straightforward emulation technique. Each X-node is modeled by a UNIX process and they will transmit messages using UNIX 'pipes'. Modeling of the kernel subprocesses within the node will be

discussed in section IIIB.

It would be possible to set up the pipes between the UNIX processes so that they form a binary tree analogous to the X-tree topology. However, the point of this exercise is to emulate the running of the X-node kernel, not the communications links. An alternative, which greatly reduces the amount of interprocess data flow, is to define a single pipe into each process. Any message is sent from one process to another by simply writing into the input pipe for the target node. The emulation network topology is then fully connected and looks like:

```
                     ----------
                    |          |-------->pipe to node 1
                    |   node   |-------->pipe to node 2
pipe ---------->|     1    |         .
                    |          |         .
                    |          |-------->pipe to node n
                     ----------
                          .
                          .
                          .
                     ----------
                    |          |-------->pipe to node 1
                    |   node   |-------->pipe to node 2
pipe ---------->|     n    |         .
                    |          |         .
                    |          |-------->pipe to node n
                     ----------
```

The simple use of pipes does not solve all of the communications problems. Each node should be continuously working on the jobs it has to do, and suspend such work only when there is a message on the pipe to be read in. Unfortunately, a read operation on an empty UNIX pipe suspends the entire process until some data is placed in the pipe by

another process. Consequently, some other mechanism is required.

This problem can be solved in part through the use of UNIX 'kill' signals (see appendix). The name 'kill' is somewhat misleading since these signals can be 'caught' by the process receiving the signal. To accomplish this, the process to be signalled performs the system call

```
int catch ();
signal (signal-number, catch);
```

and continues to execute its other tasks.

Then, when another process sends that signal-number to the process with the system call

```
kill (process-number, signal-number),
```

the function catch() will be asynchronously called in the signalled process. This call is like any other in that on exit from catch(), the signalled process will continue with the code that was being executed when the 'kill' signal was received.

One more aspect of catching signals is very important. Once a signal is caught, the 'signal' system call must be re-issued to catch the next signal. Our communication system code then looks like

```
to send message:
        write into pipe
        send signal (system call 'kill')

to receive message:
        initialize with
            "signal (signal-number, catch)"
        ...
        function CATCH:
                awaken subprocess to read pipe
                return
        ...
        subprocess INPUT:
            do forever
                read and process message
                re-issue "signal" system call
                go back to sleep
```

This is still not the final solution. Since any number of processes can write into the pipe at virtually the same instant, a dangerous condition occurs. If two processes send messages into the pipe in quick succession, the receiving process will not be able to re-issue the signal "catch" before the second signal arrives. If this happens, UNIX will abort the receiving process, which is clearly unacceptable.

To solve this problem, the X-tree node is split into two UNIX processes. The primary one, called PROC, does most of the work and operates as before. However, other processes do not write directly into its pipe. Instead, an intervening process (called COMM) receives messages from other nodes and sends them to the PROC process one at a time using a simple hand-shaking protocol:

```
    ------> PROC gets to catch signal from COMM
    !       PROC signals COMM that it is ready
    !           for message
    !       COMM gets message from its pipe
    !       COMM puts message into private pipe to PROC
    !               and sends signal to PROC
    !       PROC receives signal from COMM then
    !               reads and processes message
    !                 !
    ------------
```

Nodes which send messages are now relieved of the necessity to send signals after writing a message into the pipe for another node. The COMM process can afford to wait when it reads an empty pipe.

The resulting final topology is

```
                              intrpt
                 ----------   signal   ----------
                 !          ! - - - - >!          !--->pipe to COMM 1
                 !   COMM   !          !   PROC   !--->pipe to COMM 2
        pipe -->!     1     !--pipe--->!    1     !        .
                 !          !          !          !        .
                 !          !< - - - - !          !--->pipe to COMM n
                 ----------    ready   ----------
                               signal
                                 .
                                 .
                                 .
                              intrpt
                 ----------   signal   ----------
                 !          ! - - - - >!          !--->pipe to COMM 1
                 !   COMM   !          !   PROC   !--->pipe to COMM 2
        pipe -->!     n     !--pipe--->!    n     !        .
                 !          !          !          !        .
                 !          !< - - - - !          !--->pipe to COMM n
                 ----------    ready   ----------
                               signal
```

Only one small detail still needs attention. It is not guaranteed that data from a pipe will be read out in the same units that they were written in. Messages will not be shuffled, but they may be received more than one at a time.

When a pipe is read, all the messages currently in the pipe are returned by that single read. Hence, we must reserve a special message separator byte. The COMM process searches for this separator byte and splits off single messages to be sent to the PROC process. This, of course will not be necessary in X-tree itself, since the end-of-message will be transmitted using an extra bit.

4.2. Subprocesses and Their Synchronization

The use of multiple UNIX processes is a satisfactory way to emulate the concurrent operation of the X-tree nodes. However, this technique will not work for modeling the coroutines running on a single node. Separate UNIX processes do not share any global address space; the coroutines in a node rely heavily on such a global data space for communication. Implementing such a data space as a disk file would be both awkward and inefficient.

This difficulty implies that multiple subprocesses must somehow be 'faked' within a single UNIX process. The UNIX system offers no direct solution to this problem. Fortunately, however, software to do just that was developed as part of the Toy Operating System [7] used at U C Berkeley to teach operating system principles. This code has been copied and used in the X-tree kernel emulator. The package is briefly described below.

A subprocess is defined as an independent execution stream together with its own separate local data stack. All external data structures and variables are shared among all the subprocesses. To implement this concept, a series of data stack storage areas are allocated, one for each subprocess. At any moment, the data stack for the currently active subprocess is found in the normal stack location (growing down from address 0177777). To activate a different subprocess, the data stack for the outgoing

subprocess is copied into its stack save area. The data stack for the new subprocess is is then copied in from its save area.

```
 -------   -------   -------   -------   -------
| hdr   | | hdr   | | hdr   | | hdr   | | hdr   |
|-------| |-------| |-------| |-------| |-------|
|       | |       | |       | |       | |       |
|       | |       | |       | |       | |       |
| stack | | stack | | stack | | stack | | stack |
| save  | | save  | | save  | | save  | | save  |
|  for  | |  for  | |  for  | |  for  | |  for  |
| subpr | | subpr | | subpr | | subpr | | subpr |
|   1   | |   2   | |   3   | |   4   | |   5   |
 -------   -------   -------   -------   -------
    |                             |
    |                             |
    | Step 2:                     | Step 1:
    | load new                    | save old
    | stack                       | stack
    |                             |
    |         |          |        |
    |         | standard |        |
    |         |          |        |
     -------->|ea    |            
              |          |
               ----------
```

It is important to note that the only private storage belonging to a subprocess is in its local data stack. All code and external or global variables are shared between them. Any needed synchronization must be handled explicitly.

Subprocesses are created in a manner analogous to the forking of UNIX processes. The call

```
int priority
spfork (priority)
```

allocates a new data stack save area and stores a copy of

the data stack in it. No new data stack is loaded over the old one. Hence there are now two subprocesses which execute from the same point. The only way to distinguish the two is by the value returned from the spfork function: the child receives the value 0 and the parent receives the value 1. Consequently the code segment

```
if (spfork(3) == 0)  child ();
```

will create a child subprocess which will enter the function child(). The parent subprocess will contiaue with the next line of code.

Execution and synchronization of subprocesses is controlled though the use of semaphores, with the P and V operations defined by Dijkstra [4]. These operations are well known and will not be described here. In this implementation, semaphores are represented by a value and a linked queue of subprocesses which are waiting for the semaphore.

Semaphores are used by the subprocesses of the kernel emulator in basically two styles. In the first style, a subprocess operates on work packets taken from a FIFO queue. The basic cycle for such a subprocess is

```
loop forever
        get a work packet from queue
        process work packet
```

As part of the operation 'get work packet', a 'P' operation is performed on a semaphore which contains the number of elements in the queue. If the queue is empty, the subpro-

cess then waits until another subprocess puts an element into the queue. The slot output drivers, for example, operate this way. They take their work from a queue(IOOUTQ) of I/O control blocks for messages which are waiting to be sent out.

The second style of use is generally called private semaphores. A private semaphore is permanently assigned to a particular subprocess. When the subprocess must wait for some reason, it does a P operation on its private semaphore. It then stays asleep until some other subprocess wakes it up by doing a V on its (the sleeping subprocess's) semaphore.

One example of this style of semaphore use is the process which reads messages from the pipe. It is called INPDISP, which stands for INPut DISPatcher. It waits on a private semaphore called IOINT. The signal from the COMM process forces the execution of the catch routine which performs a V on IOINT, thus waking the IOINT subprocess.

The subprocesses for the kernel and hardware emulation were written 'on top of' these synchronization primitives. In order to keep the kernel code separate from the subprocess implementation code, almost no changes were made in copying the subprocess package from the Toy Operating System.

One change was needed, however, because the P and V operations must be indivisible in order to be valid. In the

Toy Operating System, there is no possibility of a real asynchronous interrupt from outside the program, so no special measures have to be taken. This is not the case in the X-tree kernel emulator. The COMM process may send the 'message in pipe' interrupt at any time. As described above, this will cause an immediate call to a subroutine which wakes up the INTDISP process. If this interrupt arrives while a low priority subprocess is doing a P or V operation, that operation would be suspended in the middle and an error may result.

To keep this from happening, it is desirable to disable the interrupts at the start of a P or V operation and reenable interrupts at the end of the interrupt. The overhead of actually disabling the interrupt would be unacceptably high, since it would involve a complicated signal protocol between the PROC and COMM processes. Instead, the interrupt is received, but the awakening of the INTDISP subprocess is postponed.

At entry to a P or V operation, a flag is set to indicate that interrupts are disabled. The CATCH routine must check this flag when an interrupt occurs; if disabled, the interrupt must be noted, but INTDISP cannot be awakened. Then, just before exit from the P or V operation, the interrupts are reenabled and, if an interrupt occurred in the meantime, INTDISP must be awakened.

The logic for this then looks like

```
subprocess INTDISP:
        loop forever
            set to catch interrupt with CATCH().
            signal COMM ready for interrupt
            wait by P(IDINT)
            read and process message

interrupt routine CATCH:
        if (intptdis)   intptreq = TRUE;
        else            V(IDINT);
        return;

P or V operation:
        intptdis = FALSE;
        .. do operation ..
        if (intptreq)  V(IDINT);
        intptdis = intptreq = FALSE;
        return;
```

4.3. Emulated Communications Hardware

The communications protocol between the kernel software and the communications controller was discussed in detail in chapter 2. How is the interaction between the slot device drivers and the channel emulated so that messages can be sent though the UNIX pipes?

The emulation of an output slot is relatively straightforward. Each time a device driver puts a new command in the SLOT COMMAND BYTE, it calls the OUTCOMM subroutine. OUTCOMM maintains a set of message assembly areas, one for each slot. When called, its functions are simple:

```
if command is READY FOR TRANSFER:
        take the bytes indicated by the ADDRESS
         and COUNT fields and add them to the
         end of the message
        increment the ADDRESS field by the COUNT
         and set the COUNT to zero
        place END OF BUFFER in the STATUS field
         and return

if command is TERMINATE:
        put the MESSAGE SEPARATOR byte at the end
         of the message
        write the message to the appropriate pipe
        clear the message area
        place END OF MESSAGE in the STATUS field
         and return
```

The INTERRUPT ADDRESS field in the slot control buffer is not used for the slot output emulation. The subroutine return address performs that function.

The emulation of the input slot controller cannot be done with a simple subroutine call. The output slot is

driven by events occurring inside the kernel software. The input slot, on the other hand, is driven primarily by external events, namely the presence of bytes and messages on the input line.

The emulation of the input slots is performed by the subprocess INPDISP, which has already been mentioned in section A of this chapter. INPDISP is implemented as a subprocess which is awakened whenever there is a message available on the input pipe. After reading the message from the pipe, INPDISP then emulates the input slot controller to transfer the message into the kernel.

Using the ADDRESS and COUNT fields of the selected slot, INPDISP transfers bytes from the message into the appropriate area in the node. When an event, such as BUFFER FILLED or END OF MESSAGE, is reached, INPDISP awakens the appropriate slot driver so that it can respond to the event. The slot drivers are run at a higher priority than INPDISP, so when awakened they execute immediately.

When INPDISP gets control back, it means that the input driver has finished processing the last event. INPDISP then continues to transfer more data or waits for another message. Hence the logic for INPDISP is:

```
do forever
    Wait for next message
    Read in message
    While more message remains
        Use ADDRESS and COUNT fields to transfer
         bytes from the message
        Adjust ADDRESS and COUNT fields
        If more message remains,
            Set STATUS = END OF BUFFER
            Awaken slot driver
    Set STATUS = END OF MESSAGE
    Awaken slot driver
```

This results in a back-and-forth action between the kernel slot driver and the INPDISP process. INPDISP generates an event and awakens the driver. The driver reacts to the event and then goes back to sleep, allowing INPDISP to continue. It is important that the driver subprocess have higher priority than INPDISP. If INPDISP had the higher priority, it would continue executing until it waited for the next message.

Here again the INTERRUPT ADDRESS field of the slot control area is not used explicitly. That function is handled by the subprocess package, which knows at what address to continue executing the driver when it wakes up. Instead, the space of the INTERRUPT ADDRESS field is used to store the address of the private semaphore which is used to wake up the driver.

5. CONCLUSIONS

5.1. Evaluation of the Kernel

To demonstrate the kernel emulation program, a simple user program was coded and executed. The program consisted of three user processes:

```
USER 0:
    Print 'Ucode 0 executing on node X.'
    Create a process on node 1 to execute USER 1.
    Start the process just created.
    Wait for a message.
    Print that a message was received and
        display the message.
    Exit.

USER 1:
    Print 'Ucode 1 executing on node X.'
    Create a process on node 2 to execute USER 2.
    Start the process just created.
    Print 'Ucode 1 about to send msg.'
    Send message 'It works!' to parent process.
    Exit.

USER 2:
    Print 'Ucode 2 executing on node X.'
    Exit.
```

The 'kernel calls' made to start processes and send messages are detailed in section 3.3.

When the emulator is started, it sets up the tree and starts the kernel in each node, then waits for a command from the terminal. The terminal operator must enter commands to create and start the first user process. Detailed instructions are included in appendix D.

The session for running the simple program above is reproduced below. Operator commands are shown in

parentheses (the parentheses are not present in the actual session).

```
SESSION                            COMMENTS
-------                            --------

% xtree
                                   invoke the emulator
command:(create 3 5 0)
                                   create process to run code
                                   segment 0 on node 3 with
                                   priority 5
*** Process created on node 3, named 3 1 ***
command:(start 3 1)
                                   start the process just created
*** Process 3 1 started on node 3 ***
Ucode 0 executing on node 3.
*** Process created on node 1, named 1 1 ***
*** Process 1 1 started on node 1 ***
Ucode 1 executing on node 1.
*** Process created on node 2, named 2 1 ***
*** Process 2 1 started on node 2 ***
Ucode 1 about to send msg.
Msg received by Ucode 0 was:
It works!
*** Process 3 1 terminated, node 3 ***
*** Process 1 1 terminated, node 1 ***
Ucode 2 executing on node 2.
*** Process 2 1 terminated, node 2 ***
```

The messages bracketed by '***' are informational messages provided by the kernel. The other messages were printed by the user program.

Note that USER 2 didn't actually execute until after USER 0 and USER 1 had finished. Since UNIX actually runs only one process at a time, there is no real parallelism; only one node at a time can run. Further, close examination of the user program reveals that USER 1 does not wait for USER 2 to finish execution, but rather to just start running. Hence, the sequence above is a correct one.

In short, this demonstration shows that the kernel design is logically correct and that the emulation techniques described herein actually work. However, the kernel functions in this version are extremely limited and much more work remains to be done before it reaches its final form.

The primary success of the kernel design is its flexibility. It is relatively simple and easy to modify, consisting essentially of a set of work stations (processes and device drivers) which retrieve and process work packets (PCBs and IORs). New features, such as those discussed in section 5.4, can be implemented in some cases by simply adding new options in existing processes or by adding new special purpose processes.

5.2. The Attempted Use of MODULA

In Aug 78, Harold Roberts completed an evaluation of the MODULA language [7,11]. He concluded that MODULA was viable language for systems programming in the X-tree project. As a result of his study, it was tentatively decided that MODULA was to be the primary programming language for X-tree.

However, some problems with the language were recognized. Two deficiencies were especially important to the development of the kernel.

(1) The lack of pointers forces programs to be written using array index notation. This results in somewhat inefficient code and occasionally awkward source text.

(2) Although MODULA provides for the explicit expression of multiple processes and synchronization, it was implemented only for bare (no underlying operating system) PDP and LSI-11's.

As a result of these difficulties, the kernel was to be designed using the MODULA language and then translated to C for testing and emulation. As work progressed, however, further problems were discovered which led eventually to abandoning MODULA altogether for the kernel.

The first problem surfaced during the design of the interface to the communications controller. Although it is

actually a single device, it acts like a multiple number of identical devices (slots). One straightforward way to interface to such a device is to have a different driver for each slot. This leads to relatively simple and understandable code since the algorithms do not have to worry about handling multiple slots simultaneously. But this approach is only practical if the device drivers can share common code and each can have a private pointer to the control area for its slot. If this is not possible, the amount of memory space needed for duplicated code for all the drivers would be prohibitive.

Unfortunately, this is impossible in MODULA. The addresses of device registers in device drivers must be declared as constants and cannot be passed as parameters to the driver when it is initiated.

The second problem arose when the algorithms for interpreting kernel messages were written. When a message is received, its contents are unknown. It is only after the first word (REQUEST TYPE) of the message has been examined that the format of the message is known. However, it is now impossible to break the message down into its component parts because MODULA is strongly typed and there is no such thing as a variant record. It is even impossible to write a non-MODULA routine to accomplish this since there is no way to invoke such a routine.

The only alternative in MODULA is to define all of the

possible messages such that only items of identical type go in the same relative positions in the messages. Unused fields in each message would have to be filled with dummy information. This is no way to minimize the communications bandwidth needed by the system.

The final problem is a result of the method of process synchronization provided by MODULA. Mutual exclusion is expressed by an 'interface module', a set of source text in which only one process may be active at a time. At first glance, this seems sufficient, but it leads to multiple copies of the same code.

Consider the implementation of various queues of IO control blocks in the kernel. All of the routines needed to manipulate these queues are the same. To achieve mutual exclusion, the routines must be enclosed in an interface module, but this would cause a process which wants to use queue A to wait if another process is using queue B. There is no way to prevent this except to provided separate interface modules, hence duplicated code, for each queue.

Of course, C is not without its own problems. It gives only lip service to abstractions and data hiding and provides no explicit mechanisms for multiple subprocesses and synchronization. But at least the language is powerful enough to allow the types of constructions necessary for the kernel in a relatively efficient fashion.

3.3. Subprocess Package Reconsidered

In section 3.2, the package which implements subprocesses was discussed. The decision not to modify this package did have some unfortunate repercusions in the emulator.

The first is an obvious duplication of some structure. The RUNUSER processes are represented by the structures of the subprocess package and by the PCB defined in the kernel. As a result, some fields in the PCB are not actually used since the values that would be there are kept in the subprocess package and are not made visible outside it. While merging the two structures would have no direct benefits, save some petty efficiencies, it would be a major step toward solving the two problems mentioned below.

One side effect of the current package is that a process can go to sleep only by doing a P operation on a semaphore. There is no direct way for one process to put another process to sleep or to perform any other kind of control. This ability would be convenient so that operations like ABORT PROCESS or HALT PROCESS could be implemented in the emulator.

Another problem caused by the subprocess package has to do with sending or receiving messages in the user data space. Since the user data space is all stack space it is moved in and out of the save area and is only in the normal

stack position while the process is actively executing. If the address of the user message buffer is stored in the IO block and used by the slot controller, the data will be transferred to or from the data stack of the controller, not of the user.

The current emulator gets around this by copying messages through a dynamic kernel buffer. Thus the user space is referenced only by the k-call subroutine executed in the user subprocess.

5.4. Future Tasks

The kernel in its present form provides an extremely limited set of services. However, the design and coding was done with some more general services in mind. Several possible improvements are discussed below.

ADD ASYNCHRONOUS MESSAGES - In the present kernel, a user process which attempts to send or receive a message is automatically put to sleep until the transmission is completed. While this is suitable for some applications, some service is needed whereby a user could issue a read and then continue processing and periodically check to see if a message has been received. This service would also allow the user to issue multiple read requests simultaneously (the IO number in the message header and IOB was included in anticipation of this service).

ADD A NODE INTERNAL CLOCK - A method of delaying a user process for a specified period is needed. The design of a clock driver was completed in MODULA but was not incorporated into the present emulator (the code is reproduced in appendix E). The emulation of a node clock might be accomplished by incrementing a counter at specified events, such as P and V operations, and at completion of each idle loop.

PERFORMANCE MEASUREMENT - It may be possible to instrument the emulator so that experiments can be performed to compare various possible ways of splitting complex user

tasks into multiple processes.

ADD PAGING AND MEMORY MANAGEMENT - The primary difficulty with adding this feature is that it will substantially increase the UNIX resources required by the emulation. Without this addition, however, no really meaningful performance measurements can be taken.

ERROR CORRECTION - The present kernel assumes that all messages are transmitted reliably. Code must be added to detect errors and retransmit messages when necessary.

TRANSPORT TO VAX - The present emulator runs only on the PDP 11/70 UNIX system. Direct transportation to VAX UNIX is not possible since the subprocess package used in the emulator makes use of the C data stack format on the 11/70. The data stack on the VAX is not the same. It may also be possible that the VAX subroutine linkage system makes it impossible to successfully save and restore the data stack as is done on the 11/70.

REFERENCES

[1] E Coffman and P Denning, Operating Systems Theory, Prentice-Hall, 1973, pp 8-11

[2] A Despain and D Patterson, "X-TREE: A Tree Structured Multi-process Computer Architecture", Conference Proceedings, 5th Annual Symposium on Computer Architecture, 1978, pp 144-151

[3] A Despain and D Patterson, "The Computer as a Component: Powerful Computer Systems from Monolithic Microprocessors", submitted to CACM, Sept 1978

[4] E Dijkstra, "Cooperating Sequential Processes", in Programming Languages (F Genuys, ed), Academic Press, 1968, pp 43-112

[5] T McCreary, "A Proposal for the Construction of MP-1", private paper, Dec 1978

[6] R Noyce, "Microelectronics", Scientific American, Vol 237, Nbr 3, pp 62-69, Sept 1977

[7] H Roberts, Master's Project Report, University of California at Berkeley, August 1978

[8] C Sequin, A Despain and D Patterson, "Communication in X-Tree, a Modular Multiprocessor System", Proceedings of ACM National Conference, Washington DC, Dec 1978

[9] P Suhler, Master's Project Report, University of California at Berkeley, June 1978

[10] D Tsichritzis and P Bernstein, Operating Systems, Academic Press, 1974, pp 23-26

[11] N Wirth, "MODULA: A Language for Modular Multiprogramming", Software Practice and Experience, Vol 7, 1977, pp 3-35

APPENDIX A - Kernel Data Structure Diagrams

Global Memory Object NAME:

```
 ----------------
| node           |--> node number of leaf at which object
|----------------|      resides
| unique id      |--> guaranteed unique within the node
 ----------------
```

Global Memory ADDRESS:

```
 ----------------
| object name    |
|----------------|
| offset         |--> offset from beginning of object
 ----------------
```

Process Control Block (PCB):

```
 --------------------------
| link field               |-->used for the FREEPCB
|--------------------------|         linked list
| full process NAME        |-->identifies the process globally
|--------------------------|
| priority                 |
|--------------------------|
| status word (psw)        |-->internal process status, such as
|--------------------------|     condition codes, etc
| external status word     |-->state of the process: IN-USE,
|--------------------------|     WAITING-FOR-IO, etc
| semaphore pointer        |-->used for emulator synchronization
|--------------------------|
| IOB list pointer         |-->pointer to the process's list of
|--------------------------|     active IOBs
| NAME of parent           |
|--------------------------|
| program counter          |--> Global ADDRESS
|--------------------------|
| stack pointer            |--> Global ADDRESS
 --------------------------
```

I/O Control Block (IOB):

```
 ----------------------------
| queue link pointer        |--->used to link IOB into queues
|---------------------------|
| owner IOB list link       |--->link for a process's list of
|---------------------------|     active IOBs
| owner's local NAME        |
|---------------------------|
| I/O control nbr           |--->used to match incoming messages
|---------------------------|     with the proper MRECV request
| I/O status                |--->status of IO: IN vs OUT,
|---------------------------|     and WAITING, BUSY,or COMPLETE
| action code               |--->encodes what msg manager should
|---------------------------|     do when I/O completes
| ptr to dynamic msg bfr    |--->= NULL if no dynamic buffer used
|---------------------------|
| msg body address          |
|---------------------------|
| message length            |
|---------------------------|
| header for message        |--->see message format
 ----------------------------
```

Slot Control Format:

```
 ------------------------------------------------------------------
|command | status | current    | current | interrupt              |
|  byte  |  byte  | transfer   |  byte   |  address               |
|        |        | address    |  count  |                        |
 ------------------------------------------------------------------
```

Standard Message Format:

message header	message body

target node number	target process NAME	IO id number	source node number	source process NAME

Header

Remote Request Formats:

REMOTE CREATE

request code = 1	process priority	process start address	IO nbr to reply to

REPLY TO REMOTE CREATE

request code = -1	full name of proc created

REMOTE START

request code = 2	full process name	IO nbr to reply to

REPLY TO REMOTE START

```
          ---------
      <---| request |
          |  code   |
          |  = -2   |
          ---------
```

```
***********************************************************************
defglob         Definitions for types and constants needed throughout
                the emulator

defint          Declarations needed at any point where interrupt
                processing must be turned off temporarily

defkstrucs      Data structure declarations like IOBs and PCBs which
                are used widely throughout the kernel

defpipes        Structures used for UNIX pipes and process-to-process
                communications

defqsem         Data definitions for the implementation of queues,
                semaphores and subprocesses

defslots        Definitions for the communications interface

s1main.c        Entry code to set up UNIX pipes and processes to
                simulate the tree

s2nodes.c       Code for the COMM process and the PROC routines
                which emulate the operation of the Xtree
                communications controller

s3boot.c        Code to initialize resident kernel data and start up
                the various subprocesses for the kernel and hardware
                simulation

s4cman.c        Kernel communications manager, including comm
                channel device drivers

s5mman.c        Kernel message manager, including the main kernel
                process

s6pman.c        Kernel user process manager, including the
                processes which emulate the running of a
                user process

s7kcalls.c      Code for the kernel service request routines which
                are called from the user processes

s8hash.c        The node central directory hash table and the
                routines for doing update and lookup operations

s9misc.c        Miscellaneous utility routine used throughout the
                kernel

sAucodes.c      Code for user programs run by the emulator

u1queues.c      Generalized code for handling all queues; queue
                handling is treated as a critical section

u2pandv.c       Implements the p() and v() synchronization operations
                on semaphores
```

```
u3subp.c        Code for maintainence and switching of subprocesses
**********************************************************************
```

```
/*****************************************************************************
        Definitions needed thoughout the emulator code.
        The NULL pointer is defined as 0177777 rather than 0 so that
          it will generate a bus error if it is followed.
        The types NODENBR is defined as distinct from normal integers
          so that it can be changed to a varying length quantity at a
          later date.
        The concept of a global NAME and global ADDRESS are also
          defined universally.
        The variable 'node' represents a globally available register
          which contains the current node number.
*****************************************************************************/
#define NULL    0177777
#define TRUE    1
#define FALSE   0
typedef int     NODENBR;
typedef struct {
        NODENBR node;
        int     id;
        }               NAME;
typedef struct {
        NAME    obj;
        char    *offset;
        }               ADDRESS;

#define NBRNODES        3
NODENBR node;
```

```
/************************************************************************
        These declarations are needed at any point where interrupt
        processing must be turned off temporarily.  There are two
        reasons to do this.
                1.  During a 'p' or 'v' operation which must be
                    indivisible
                2.  During a print or similar operation during
                    which the data stack will be too large to fit
                    in the save area.
************************************************************************/
int        intptdis,        intptreq;
#define DISABLE intptdis=1;
#define ENABLE  if (intptreq) v(ioint);  intptdis=intptreq=0;
struct sem      *ioint;
```

```
/**********************************************************************
        Data structure declarations which are used widely throughout
        the kernel.
**********************************************************************/

/*********************************************************
Pointers to semaphores and FIFO queues are used
throughout the kernel, but the structures
themselves are manipulated only by routines in
ul.queues and ul.pandv.
*********************************************************/
struct sem {
        /* It's a secret */
        };
struct qheadr {
        /* It's a secret */
        };

/*********************************************************
What the header on all messages looks like.
*********************************************************/
struct hdbfr {
        NODENBR tonode;
        NAME    toproc;
        int     tofonbr;
        NODENBR fromnode;
        NAME    fromproc;
        };

/*********************************************************
Dynamically available message buffers.  Unused
ones are linked in a queue.
*********************************************************/
#define KMSGSIZ 80
#define NBRKMSG 10
struct kmsg {
        struct kmsg     *kmlink;
        char    kbfr[KMSGSIZ];
        }               kmsgarr[NBRKMSG];
struct qheadr   *freekq;

/*********************************************************
Formats for the messages which the kernel deals
with directly.
*********************************************************/
struct remcrt  {                        /* Remote user create */
        int     rreq;                   /*  = PCREATE  */
        int     rprio;
        ADDRESS rstart;
        int     retcio;
        };
struct remcrtb  {                       /* Response to remcrt */
        int     rreq;                   /*  = -PCREATE  */
        NAME    newchild;
        };
struct remstr  {                        /* Remote user start */
```

```
        int     rreq;           /*  = PSTART   */
        NAME    rproc;
        int     retsio;
        };
struct remstrb  {               /* Response to remstr */
        int     rreq;           /*  = -PSTART   */
        };
/*****************************************************
Codes for the emplemented kernel services.
*****************************************************/
#define PCREATE 01
#define PSTART  02
#define SEXIT   11
#define MSEND   21
#define MRECV   22

/*****************************************************
I/O control blocks for defining and routing
messages.
*****************************************************/
#define NBRIOBS 20
struct ioblock {
        struct ioblock  *ioblink;
        struct ioblock  *ownlink;
        struct ioblock  *owner;
        int      relionbr;
        int      iostat;
        int      ioaction;
        struct kmsg     *kmptr;
        ADDRESS msgaddr;
        int      msglng;
        struct hdbfr    msghdr;
        }                       iobarr[NBRIOBS];
struct qheadr   *freeiobs,      /* queue of unused blocks */
                *actionq,       /* queue of completed messages */
                *iooutq;        /* queue of messages to be sent */
struct ioblock  *kiobs;
/*  I/O status bits  */
#define IOIN            0200
#define IOOUT           0100
#define IOWAIT          0004
#define IOBUSY          0002
#define IODONE          0001
/*  Action codes (used at completion of I/O)  */
#define UNSPECIF        00
#define DISCARD         01
#define ACTUSER         02

/*****************************************************
Process control blocks - one for each possible
active process in the node.
*****************************************************/
#define NBRUSERS        5
struct pcblock {
        struct pcblock  *pclink;
        NAME    pcid;
```

```
        int     pcprio;
        int     psw;
        int     extstatus;
        struct sem      *privsem;
        struct inblock  *pciobs;
        NAME    parent;
        ADDRESS pc;
        ADDRESS pstk;
        }               pcbarr[NBRUSERS+1];
struct pcblock  *freepcbs;      /* queue of unused PCBs */
/*  External status bits  */
#define INUSE           0001
#define STOPPED         0002
#define DELAYED         0004
#define TORLOCKED       0010

/*************************************************
Codes for hash table updates and requests
*************************************************/
#define HDEL    00
#define HNODF   01
#define HPCB    02
```

```
/**********************************************************************
Structures used for UNIX pipes and process-to-process communication.
**********************************************************************/
/*******************************************
File descriptor for a UNIX pipe
*******************************************/
struct  fildes  {
        int     rddes;
        int     wtdes;
        };

/*******************************************
Table of entries which hold, for each node, the
id numbers of the UNIX processes and input pipes
which represent the node.
*******************************************/
struct  unixid  {
        int     commid;
        struct fildes   commpipe;
        int     procid;
        struct fildes   procpipe;
        }                       nodearr[NBRNODES], *n;

/********************************************
Auxiliary definitions used:
 a. A message separator character needed because
    UNIX may return more than one message at a time
    from a pipe read.
 b. Number of UNIX signal sent from 'proc' process
    to 'comm' process to say it is ready for the
    next message.
 c. Number of UNIX signal sent from 'comm' process
    to 'proc' process to say another message is in
    the pipe.
********************************************/
#define MSGEND  '@'
#define ENBLSIG 15
#define INTSIG  15

/********************************************
Buffer used to read data from the input pipes
into each process
********************************************/
#define CBFRSIZ 100
struct cmsgbfr {
        char    *nextmsg;
        char    *nextchar;
        char    bfr[CBFRSIZ];
        }                       mb;
```

```
/*********************************************************************
        Data definitions for the implementation of queues,
        semaphores and subprocesses.
*********************************************************************/
#define SEMMAX          100     /*  Number of available semaphores  */
#define STACKMAX        150     /*  Size of stack save area for a
                                        subprocess              */
#define SPMAX           16      /*  Number of subprocess areas  */
#define QCNT            20      /*  Number of available queues  */
#define boolean         int
#define true            1
#define false           0
struct stateform {              /* Save area for a subprocess stack */
        int     stkptr;
        int     stack[STACKMAX];
        };
struct sp {                     /* Subprocess environment */
        struct sp       *splink;
        int     priority;
        struct stateform        state;
        };
struct sem {                    /* Semaphore: value & queue of waiting
                                        sp's */
        int     value;
        struct sp       *qhead;
        };
struct qform {                  /* A queue element; only the top
                                        pointer is used */
        struct qform    *qlink;
        /* plus the oulect in this queue */
        };
struct qheadr {                 /* Header element for a FIFO queue */
        struct sem      *count;
        struct sem      *mutex;
        struct qform    *head;
        struct qform    *tail;
        };
```

```
/*****************************************************************
        Definitions for the communications interface.
*****************************************************************/

#define NBRSLOTS        4

/**********************************************
The slot interface definition
**********************************************/
struct slotctl {
        int      status;
        ADDRESS addr;
        int      bcnt;
        struct sem      *sltint;
        }               inslot[NBRSLOTS], outslot[NBRSLOTS];
/*  Command and status bits  */
#define READY   0200
#define TERMIN  0100
#define FOBUF   0004
#define FOMSG   0002
#define ERROR   0001

/**********************************************
Dedicated buffers to receive input message
headers
**********************************************/
struct hdbfr    hb[NBRSLOTS];

/**********************************************
Buffers in which to build output messages
**********************************************/
#define COUTLNG 80
struct coutmsg {
        int     coutlng;
        char    coutbuf[COUTLNG];
        }               coutarr[NBRSLOTS];
```

```
#include <stdio.h>
#include "defgloh"
#include "defpipes"

/*******************************************************************
        This is the entry code which starts up in the original
        UNIX process.  It first sets up the pipes necessary for
        process-to-process communication.  Then it 'forks' off
        a 'comm' process for each X-tree node.  Finally it goes
        into a loop in which it accepts commands from the terminal
        to create or start processes in the nodes.

        Each 'comm' process "forks" off the 'proc' process for
        its node.  The 'comm' process then calls the routine
        'commun', where it stays forever.  Similarly, the 'proc'
        process calls the routine 'processor' and stays there
        forever.
*******************************************************************/
main ()  {
for (n=nodearr; n<nodearr+NBRNODES; n++)  {  /* set up pipes */
        pipe (&n->commpipe);
        pipe (&n->procpipe);
        };
for (n=nodearr; n<nodearr+NBRNODES; n++)  {  /* start up processes */
        node = (n - nodearr) + 1;
        if (!(n->commid = fork()))  {
                /* Only the 'comm' process gets here */
                n->commid = getpid ();
                if (!(n->procid = fork()))
                        /* Only the 'proc' process gets here */
                        processor ();
                else
                        /* The 'comm' process continues here */
                        commun ();
                }
        /* The main process continues here */
        }
operator ();
}

/*******************************************************************
*       Operator communication:  This is a kludge to allow the
*       functions of the kernel to be tested.  The standard test
*       sequence is:
*               create 3 5 0
*               start 3 1
*       This starts up a process on node 3 which executes
*       user code routine 0.
*******************************************************************/
#include "defkstrucs"
static operator ()  {
char what[10];
int nbyte, match;
struct remcrt *cptr;
struct remstr *sptr;
```

```
struct { struct hdbfr mhd;
         char mbody[80];
       }            msg;

while ((match=(printf("command:"),scanf("%s",what))) != EOF)  {
        int tnode, prio, pcnode, pcid, pcoffs;
        if (match != 1)  continue;
        if (!strcmp (what, "create"))  {

                /*** Send remote process creation message ***/
                if ((match=scanf("%d%d%d", &tnode, &prio,
                    &pcid)) != 3)
                        printf ("Cmd error at %d args\n", match);
                if (tnode<1)  tnode=1;
                if (tnode>NBRNODES)  tnode=NBRNODES;
                pcnode = tnode;
                pcoffs = 0;

                /*** Fill in message header ***/
                msg.mhd.tonode = tnode;
                nilname (&msg.mhd.toproc);
                msg.mhd.tofonbr = msg.mhd.fromnode = 0;
                nilname (&msg.mhd.fromproc);

                /*** Fill in message body ***/
                cptr = msg.mbody;
                cptr->rreq = PCREATE;
                cptr->rprio = prio;
                cptr->rstart.obj.node = pcnode;
                cptr->rstart.obj.id = pcid;
                cptr->rstart.offset = pcoffs;
                cptr->retcio = 0;

                /*** Put message separator on the end of message ***/
                nbyte = sizeof(*cptr);
                msg.mbody[nbyte++] = MSGEND;

                /*** Send message to the appropriate node ***/
                nbyte += sizeof (msg.mhd);
                vermsg (&msg,nbyte);
                write (nodearr[tnode-1].commpipe.wtdes, &msg, nbyte);
                }
        else if (!strcmp (what, "start"))  {

                /*** Send remote process start message ***/
                if ((match=scanf("%d%d",
                    &pcnode, &pcid)) != 2)
                        printf ("Cmd error at %d args\n", match);
                tnode = pcnode;
                if (tnode<1)  tnode=1;
                if (tnode>NBRNODES)  tnode=NBRNODES;

                /*** Fill in message header ***/
                msg.mhd.tonode = tnode;
                nilname (&msg.mhd.toproc);
                msg.mhd.tofonbr = msg.mhd.fromnode = 0;
```

```
                        nilname (&msg.mhd.fromproc);

                        /*** Fill in message body ***/
                        sptr = msg.mbody;
                        sptr->rreq = PSTART;
                        sptr->rproc.node = pcnode;
                        sptr->rproc.id = pcid;
                        sptr->retsio = 0;

                        /*** Put message separator on the end of message ***/
                        nbyte = sizeof(*sptr);
                        msg.mbody[nbyte++] = MSGEND;

                        /*** Send message to the appropriate node ***/
                        nbyte += sizeof (msg.mhd);
                        vermsg (&msg,nbyte);
                        write (nodearr[tnode-1].commpipe.wtdes, &msg, nbyte);
                        }
                else printf ("Invalid command\n");
                }
        }
```

```
/*****************************************************************
*       This source file contains the routines which together
*       vaguely look like the communication controller.
*****************************************************************/
#include <stdio.h>
#include "defglob"
#include "defpipes"
#include "defkstrucs"
#include "defint"
#include "defslots"

int     enabled = FALSE;          /* = TRUE iff the 'proc' process is
                                     ready to receive the next
                                     message and interrupt */

/*****************************************************************
*       COMMUN - The infinite loop for the 'comm' process.  It
*       reads messages from other nodes, assigns a slot address
*       to them, and sends them to the 'proc' process.
*****************************************************************/
commun ()  {
int     enable(), lng;
char    sa = NBRSLOTS-1;

signal (ENBLSIG,enable);          /* Set up to catch the first enable
                                     signal from 'proc' */
mb.nextmsg = mb.nextchar = mb.bfr;       /* Initialize input buffer */
while (TRUE)   {
        getmsg (&mb);             /* Get next message from pipe */
        while (!enabled)  sleep (1);    /* Wait till 'proc' ready for
                                           interrupts */
        enabled = FALSE;
        signal (ENBLSIG,enable);        /* Set to catch next signal */
        sa = (sa+1) % NBRSLOTS;         /* Rotate to next slot */
                /** Write message, with target slot number added
                    at the front, to 'proc'; then send interrupt
                    signal  **/
        write ((n->procpipe).wtdes, &sa, 1);
        lng = mb.nextmsg - mb.bfr - 1;
        write ((n->procpipe).wtdes, mb.bfr, lng);
        kill (n->procid, INTSIG);
        }
}

        /*********************************************************
        *  GETMSG - called only from COMMUN.  Data from the pipe
        *  is viewed by UNIX as a stream of bytes, so we are not
        *  guaranteed to get just one message from the pipe at each
        *  read.  GETMSG performs the pipe read and splits the
        *  stream into individual messages, returning them one at a
        *  time.  Note that this means that a message separator
        *  character must be reserved.
        *********************************************************/
static getmsg (bp)
struct  cmsgbfr *bp;    {
char    *i, *j;
```

```
int     mxlng, lng;

for (i=(bp->bfr), j=(bp->nextmsg); j<(bp->nextchar); i++,j++)
        *i = *j;        /* Move remaining chars to front of buffer */
bp->nextmsg = bp->bfr;  bp->nextchar = i;
while (TRUE)    {

                /*** if end of message found, return ***/
        for (i=(bp->bfr); i<(bp->nextchar); i++)        {
                if (*i == MSGEND)       {
                        bp->nextmsg = i+1;
                        return;
                        }
                }

                /** read more characters from the pipe **/
        mxlng = (bp->bfr) + CBFRSIZ - (bp->nextchar);
        if (mxlng<=0)  printf ("comm bfr full but no MSGEND found\n");
        while ((lng = read (n->commpipe.rddes, bp->nextchar, mxlng))
                        < 0);
#ifdef DEBUG
        printf ("%d chars received as \n", lng);
        show (bp->bfr, lng);
#endif
        (bp->nextchar) += lng;

}

/***********************************************************************
*       ENABLE - This routine is called asyncronously when the
*       'enable' signal is sent from the 'proc' process.  The UNIX
*       system is told to do this by the call to function 'signal' in
*       COMMUN above.
***********************************************************************/
static enable ()        {
enabled = TRUE;
}


/***********************************************************************
*       INPDISP - This "input dispatcher" runs as an independent
*       subprocess in 'proc'.  It is awakened whenever the INTSIG
*       signal is received from the 'comm' process.
***********************************************************************/
inpdisp ()  {
int     catch ();
int     lng, i;
char    *j;
while (TRUE)  {
                /* Set to have the routine 'catch' executed when the
                   next INTSIG signal is received.  Then signal 'comm'
                   that we are ready for the next message and wait
                   for it.  */
        signal (INTSIG, catch);
        kill ((n->commid), ENBLSTG);
        p(ioint);
```

```
		/* Read the message and pull off the slot number */
	lng = read ((n->procpipe).rddes, mb.bfr, CBFRSIZ);
#ifdef DEBUG
	DISABLE
	printf ("message received in processor %d, length %d:\n",
		node, lng);
	show (mb.bfr, lng);
	printf ("%s\n", mb.bfr+1);
	ENABLE
#endif
	i = mb.bfr[0];
	if (inslot[i].status != READY)  error ("Input on busy slot.");

		/* Fill up the buffer defined by the slot control address
			and word count, then awaken the kernel driver with
			an end-of-buffer indication */
		/* Note that this subprocess must have lower priority than
			driver subprocess in order that the driver will get
			control when the 'v' is performed */
	for (j=mb.bfr+1; j<mb.bfr+lng; j++)  {
		while (inslot[i].bcnt == 0)  {
			inslot[i].status = EOBUF;
			v(inslot[i].sltint);
			}
		*(inslot[i].addr.offset++) = *j;
		inslot[i].bcnt--;
		}

		/* At the end of message, awaken the driver with an
			end-of-message indication */
	inslot[i].status = EOMSG;
	v(inslot[i].sltint);
	}
}

/**********************************************************************
*	OUTCOMM - This output communications routine is called by
*	the kernel output slot drivers whenever they execute a
*	'ready to send buffer' or terminate message command.
*	The full message is assembled in a buffer, then written
*	onto the appropriate pipe.
**********************************************************************/
outcomm (slot, cbf)
struct slotctl *slot;
struct coutmsg *cbf;	{
int dest, *dptr;

if (slot->status == TERMIN)  {
		/* Tack EOM char at end of message and verify the
			message for valid target and contents */
	cbf->coutbuf[(cbf->coutlng)++] = MSGEND;
	vermsg (cbf->coutbuf, cbf->coutlng);
	dptr = cbf->coutbuf;
	dest = *dptr;
	if (dest<0 || dest>NBRSLOTS)
```

```
                        error ("Bad destination.");
                if (dest == 0)  {
                                /* Node 0 indicates message to operator.  Write
                                    to terminal in a kludgy, but readable, way */
#ifdef DEBUG
                        printf ("message to terminal.");
                        show (cbf->coutbuf, cbf->coutlng);
#endif
                        }
                else
                        write (nodearr[dest-1].commpipe.wtdes,
                                        cbf->coutbuf, cbf->coutlng);
                    /* Return end-of-message status */
                cbf->coutlng = 0;
                slot->status = EOMSG;
                return;
                }
else if (slot->status == READY)  {
                while (TPUF)  {
                                /* Move bytes from message into the assembly
                                    area until message buffer is empty */
                        if (slot->bcnt <= 0)  {
                                slot->status = EOBUF;
                                return;
                                }
                        (cbf->coutbuf)[(cbf->coutlng)++] =
                                        cfetch (&(slot->addr));
                        (slot->addr).offset++;
                        (slot->bcnt)--;
                        if (cbf->coutlng >= COUTLNG)
                                error ("Out msg too long.");
                        }
                }
}

/*****************************************************************************
*       VERMSG - verify that a message area has a terminator at the
*       end and nowhere else
*****************************************************************************/
vermsg (c,l)
char *c;  int l;  {
char *end;
end = c+l-1;
while (c<end)
        if (*(c++) == MSGEND)
                error ("Msg contains terminator.");
if (*c != MSGEND)
        error ("No terminator on msg.");
}

/*****************************************************************************
*       CATCH - Routine called when INTSIG received from 'comm'
*       process.  When this happens, INTDISP must be awakened
*       by 'v'ing the IOINT semaphore.  However, if interrupts
*       have been temporarily disabled for some reason, the
```

```
*       interrupt is simply noted and the 'v' action is then
*       taken when interrupts are next re-enabled
*******************************************************************/
static catch () {
if (intptdis)
        intptreq = TRUE;
else
        v(joint);
}
```

```
#include <stdio.h>
#include "defglob"
#include "defkstrucs"
#include "defslots"
#include "defint"

/*********************************************************************
*       PROCESSOR - Initial code for each 'proc' process.  It calls
*       various data initialization routines and then activates
*       the subprocesses for the kernel and hardware simulation:
*         - the Input Dipatcher subprocess (see s2nodes)
*         - an Input Driver for each slot (see s4cman)
*         - an Output Driver for each slot (see s4cman)
*         - the Kernel Message handler (see s5mman)
*         - a Runuser for each potential active user (see s6pman)
*       This then becomes the cpu idling process.
*********************************************************************/
processor ()  {
int     i;

initio ();
initusr ();
initmman ();
inithsh ();

if (!spfork(50))  inpdisp ();
for (i=0; i<NBRSLOTS; i++)  {
        if (!spfork(60))  inpdriver (i);
        }
for (i=0; i<NBRSLOTS; i++)  {
        if (!spfork(60))  outdriver (i);
        }
if (!spfork(30))  kmproc ();
for (i=1; i<NBRUSERS+1; i++)
        if (!spfork(1))         user(i);
while (TRUE)  {
        ENABLE
        sleep (0);
        }
}
```

```
/*****************************************************************
*       CMAN - Communication Management
*****************************************************************/
#include <stdio.h>
#include "defglob"
#include "defkstrucs"
#include "defint"
#include "defslots"

/*****************************************************************
*       INITIO - Initializes the data structures for communications
*****************************************************************/
initio ()  {
int i;
ioint = newsem(0);
for (i=0; i<NBRSLOTS; i++)  {
        inslot[i].status = 0;
        inslot[i].sltint = newsem(0);
        outslot[i].status = 0;
        }
intptdis = intptreq = FALSE;
iooutq = newq ();
for (i=0; i<NBRSLOTS; i++)
        coutarr[i].couting = 0;
}

/*****************************************************************
*       INPDRIVER - Device driver for input slots.  This reentrant
*       code is shared by each slot, with separate data stacks and
*       control areas for each slot.
*****************************************************************/
inpdriver (i)  int i;  {
struct slotctl *slot;    /* Pointers to slot control area */
struct sem *intpt;       /*           semaphore to wait on INPDISP */
struct hdbfr *inh;       /*           dedicated header input buffer */
struct ioblock *iob, *findiob (); /*  current I/O control block */

slot = &inslot[i];
intpt = slot->sltint;
inh = &hb[i];

while (TRUE)  {
        nilname (&(slot->addr),obj);     /* set up to receive msg header */
        (slot->addr).offset = inh;       /* then wait for it */
        slot->hcnt = sizeof (hb[0]);
        slot->status = READY;
        p(intpt);
        if ((slot->status) &= EOBUF)  error ("Message only has hdr.");

          /* Find or create an i/o block for this message */
        if ((iob=findiob(inh,IOIN))==NULL)  error ("No iob match");
        iob->instat = (IOIN | IOBUSY);
          /* Copy the header into the i/o block for later use  */
        bytecopy (&iob->msghdr, inh, sizeof(*inh));

          /* Set up to receive the msg body, then wait for it */
```

```
        addrcopy (&(slot->addr), &(iob->msgaddr));
        slot->bcnt = iob->msglng;
        slot->status = READY;
        p(intpt);
        if ((slot->status) == FUMSG)  error ("Message is too long.");

          /* Put the actual message length in the i/o block, then
             put the IOB on the queue of completed messages awaiting
             action  */
        iob->msglng -= slot->bcnt;
        iob->iostat = (iob->iostat & ~IOBUSY) | IODONE;
        put (actionq, &iob);
        }
}

/************************************************************************
*       FINDIOB - Locates the IOB that a particular input message
*       is to be matched with.  If none already exists, one is
*       allocated for it.  If needed, a dynamic message buffer is
*       also allocated for it.
************************************************************************/
struct ioblock *findiob (hdr,type)
struct hdbfr *hdr;
int type;               {
struct ioblock *newiob ();
struct kmsg *newkbf ();
struct ioblock *i;
NAME *n;
int ktarget;
struct pcblock *p;

   /* Locate the process that this message is directed to */
n = &(hdr->toproc);
if (n->id==0)   {
        ktarget = TRUE;
        i = kiobs;
        }
else    {
        ktarget = FALSE;
        if (hfind (&hdr->toproc, &p) != HPCB)
                return (NULL);
        i = p->pciobs;
        }

  /* Search list of the process IOB's for a match */
while (i != NULL)  {
        /* A match must have same io nbr and direction (in or out) */
        if ((hdr->toionbr==i->relionbr) && ((i->iostat)&type))  {
                if (addrnil(&i->msgaddr))  {
                        /* allocate a msg buffer if needed */
                        i->kmptr = newkbf ();
                        (i->msgaddr).offset = (i->kmptr)->kbfr;
                        i->msglng = KMSGSIZ;
                        }
                return (i);
                }
```

```
		i = i->ownlink;
		}

	/* If no match was found, allocate an IOB and a buffer */
	if (ktarget)	{
		i = newiob ();
		i->iostat = type;
		i->kmptr = newkbf ();
		i->msgaddr.offset = (i->kmptr)->kbfr;
		i->msglng = KMSGSIZ;
		return (i);
		}
	else
		return (NULL);
	}

/**********************************************************************
*	OUTDRIVER - Device driver for output slots.  This reentrant
*	code is shared by each slot, with separate data stacks and
*	control areas for each slot.
**********************************************************************/
outdriver (i)  int i;	{
struct slotctl *slot;
struct coutmsg *cbf;
struct ioblock *iob;
slot = &outslot[i];  cbf = &coutarr[i];

while (TRUE)	{
	iob = get (iouto);	/* get next IOB from queue of IOBs
				awaiting output */
	if (iob->iostat != (IOOUT | IOWAIT))
		error ("Bad iob in outdriver.");
	iob->iostat = (IOOUT | IOBUSY);

	  /*  Set up slot addr and word count to output
	      message header from IOB */
	niladdr (&(slot->addr));
	slot->addr.offset = &iob->msghdr;
	slot->bcnt = sizeof (hb[0]);
	slot->status = READY;
	outcomm (slot, cbf);
	if (slot->status != FOBUF)
		error ("Failure sending header.");

	  /* Send message body after setting up slot addr and count */
	addrcopy (&(slot->addr), &(iob->msgaddr));
	slot->bcnt = iob->msglng;
	slot->status = READY;
	outcomm (slot, cbf);
	if  (slot->status != FOBUF)
		error ("Failure sending body.");

	  /* Terminate message */
	slot->status = TERMIN;
	outcomm (slot, cbf);
	if (slot->status != FOMSG)
```

```
                error ("Failure terminating msg");

            /* Send TUR back to message handler for internal
                termination actions  */
        ioh->iostat = (IODUT | IODONE);
        put (actionq, &iob);
        }
}
```

```
/************************************************************************
*          MMAN - Message Management
************************************************************************/
#include <stdio.h>
#include "defglob"
#include "defkstrucs"
#include "defint"

/************************************************************************
*          INITMMAN - Initializes message management storage structures
************************************************************************/
initmman ()  {
struct qform *h;
int i;
freekq = newq ();                       /*  Put all the dynamic message */
for (i=0  i<NBRKMSG; i++)   {           /*  buffers in the free queue */
        h = &kmsgarr[i];
        put (freekq, &h);
        }
freeiobs = newq ();                     /* Put all the IOBs in a free queue */
for (i=0; i<NBRIOBS; i++)   {
        h = &iobarr[i];
        put (freeiobs, &h);
        }
actionq = newq ();                      /* Set up an initially empty queue of
                                           completed messages */
kiobs = NULL;
}

/************************************************************************
*          KMPROC - the Kernels Message Handling process
************************************************************************/
kmproc ()  {
struct iobloack *i;
struct kmsg *k;
struct hdbfr *h;
int     *req;

while (TRUE)  {

          /* Get a completed message from the action queue */
        i = get (actionq);
        k = i->kmptr;
        h = &(i->msghdr);
        if (!(i->iostat & IODONE))
                error ("Iob in actionq not DONE.");

          /* Update the hash table of known processes */
        if (i->iostat & IOIN  &&  (h->fromproc).id != 0)
                hupdate(&h->fromproc,HNODE,h->fromnode);

        if ((i->iostat & IOIN) && (i->ioaction == UNSPECIF))    {
                  /* Message is a remote request to this kernel */
                if (i->msghdr.toproc.id != 0)
                        error ("UNSPEC iob not for kernel.");
#ifdef DEBUG 1
```

```
                DISABLE
                printf ("Kmsg of length %d\n", i->msglng);
                show (k->khfr, i->msglng);
                printf ("Header was \n");
                show (h, sizeof(*h));
                ENABLE
#endif
                req = (i->msgaddr.offset);
                switch (*req)  {
                    case PCREATE:  {
                        struct remcrt *rin;
                        struct remcrtb *rout;
                        struct pcblock *newu;
                        rin = rout = req;

                          /* allocate a new PCB */
                        newu = kcrtusr (&rin->rstart, rin->rprio, &h->fromproc);

                          /* set up the TOB for the reply message */
                        i->relionbr = rin->retcio;
                        i->iostat = TOOUT ! TOWAIT;
                        i->ioaction = DISCARD;
                        i->msglng = sizeof (*rout);

                          /* set up the header (send to requesting process) */
                        h->tonode = h->fromnode;
                        namecopy (&(h->toproc), &(h->fromproc));
                        h->toionbr = rin->retcio;
                        h->fromnode = node;
                        namecopy (&(h->fromproc), &(newu->pcid));

                          /* fill in the message and put in the output queue */
                        rout->rreq = -PCREATE;
                        namecopy (&rout->newchld, &newu->pcid);
                        put (iooutq, &i);
                        continue;
                        }
                    case PSTART:  {
                        struct remstr *rin;
                        struct remstrb *rout;
                        struct pcblock *p;
                        rin = rout = req;

                          /* locate the process to be started */
                        if (hfind(&rin->rproc,&p) == HPCB)
                                kstrtusr (p);
                        else
                                error ("No process for PSTART");

                          /* set up the TOB for the reply message */
                        i->relionbr = rin->retsio;
                        i->iostat = TOOUT ! TOWAIT;
                        i->ioaction = DISCARD;
                        i->msglng = sizeof (*rout);

                          /* set up the header (send to requesting process) */
```

```
                        h->tonode = h->fromnode;
                        namecopy (&(h->toproc), &(h->fromproc));
                        h->toionbr = rin->retsio;
                        h->fromnode = node;
                        nilname (&h->fromproc);

                          /* fill in the message and put in the output queue */
                        rout->rreq = -PSTART;
                        put (iooutq, &i);
                        continue;
                        }
                        break;
                    default:
                        error ("Bad remote kernel request.");
                    }
                }
        if (i->ioaction & ACTUSER)
                /* Resume running of the owner of the message */
            v ((i->owner)->privsem);
        if (i->ioaction & DISCARD)
                /* Release the IOB and message buffer */
            reliob (i);
        }
}

/*********************************************************************
*       NEWIOB - gets a new IOB off the free queue and initializes
*       it with zeroes and NULLs
*********************************************************************/
struct ioblock *newiob ()  {
struct ioblock *i;
i = get (freeiobs);
i->ownlink = NULL;
i->owner = NULL;
i->relionbr = 0;
i->iostat = 0;
i->ioaction = 0;
i->kmptr = NULL;
niladdr (&(i->msgaddr));
i->msglng = 0;
nilhdr (&(i->msghdr));
return (i);
}

/*********************************************************************
*       NEWKBF - gets a new dynamic message buffer off the free
*       queue and initializes it with zeroes and NULLs
*********************************************************************/
struct kmsg *newkbf ()  {
struct kmsg *km;
char *c;
km = get (freekq);
c = km->kbfr;
while (c<(km->kbfr)+KMSGSIZ)  *(c++) = '\0';
return (km);
```

```
}

/**********************************************************************
*         RELIOB - Release an IOB back to the free queue
**********************************************************************/
reliob (iob)    struct ioblock *iob;  {

  /* if it's currently in active use, let the action complete first */
if (iob->iostat & IOBUSY  ||  iob->ioblink != NULL)  {
          iob->ioaction = DISCARD;
          iob->owner = NULL;
          return;
          }
else      {
          if (iob->owner != NULL)  {
                    /* Delink it from its owner */
                    struct ioblock *i;
                    i = (iob->owner)->pciobs;
                    if (i == iob)  (iob->owner)->pciobs = iob->ownlink;
                    else      {
                              while (i != NULL  &&  i->ownlink != iob)
                                        i = i->ownlink;
                              if (i != NULL)  i->ownlink = iob->ownlink;
                              }
                    iob->owner = NULL;
                    }
          iob->ownlink = NULL;
            /* If it has a dynamic buffer, release that too */
          if (iob->kmptr != NULL)  put (freekq, &iob->kmptr);
            /* Put IOB on free queue */
          put (freeiobs, &iob);
          }
 }
```

```
/**********************************************************************
*        PMAN - Process Management
**********************************************************************/
#include <stdio.h>
#include "defglob"
#include "defkstrucs"
#include "defint"


/**********************************************************************
*        KCRTUSR - Create a user process
**********************************************************************/
struct pcblock *kcrtusr (start, prio, parent)
ADDRESS *start;  int prio;  NAME *parent;  {
struct pcblock *p;  int f;

   /*  Get an unused process control block */
p = get (freepcbs);

   /*  Fill in the appropriate values  */
p->pcprio = prio;
p->psw = 0;
p->extstatus = INUSE | STOPPED;
p->pcjobs = NULL;
namecopy (&p->parent, parent);
addrcopy (&p->pc, start);
(p->pcid).node = node;

   /* Select a unique identifier for the 'id' part of the
      full process name */
do      (p->pcid).id = nextid();
while (hfind(&p->pcid, &f) != HDFL);

   /* Add the new process name to the hash table */
hupdate (&p->pcid, HPCB, p);
DISABLE
printf ("*** Process created on node %d, named %d %d ***\n",
        node, p->pcid.node, p->pcid.id);
ENABLE
return (p);
}

/**********************************************************************
*        NEXTID - Generates a new id nbr to be tested for uniqueness
**********************************************************************/
static nextid()  {
static int lastid;
lastid = lastid % 30000   +   1;
return (lastid);
}

/**********************************************************************
*        KSTRTUSR - Starts up a selected user process by 'v'ing its
*        private semaphore to awaken the kernel subprocess running it
**********************************************************************/
kstrtusr (p)    struct pcblock *p;  {
```

```
p->extstatus &= ~STOPPED;
v (p->privsem);
return;
}

/*************************************************************************
*       USRDSTRY - Destroy a user process once it has completed
*************************************************************************/
static usrdstry (ps)    struct pcblock *ps;  {
struct ioblock *iob, *iobn;

  /* Delete its entry from the hash table */
hupdate (&ps->pcid, HUFL, 0);

  /* Release all its IOBs */
for (iob=ps->pciobs; iob!=NULL; iob=iobn)  {
        iobn = iob->ioblink;
        iob->owner = NULL;
        reliob (iob);
        }

  /* Put the PCB back on the free queue */
ps->extstatus = 0;
nilname (&ps->pcid);
put (freepcbs, &ps);
}

/*************************************************************************/
/*  Table of pointers to predefined user code segments  */
#define NBRUCODES       3
static int      (*ucode[NBRUCODES]) ();

/*************************************************************************
*       INITUSR - Initialize data structures pertaining to PCBs and
*       user code segments
*************************************************************************/
initusr ()  {
int     ucode0 (), ucode1 (), ucode2 ();
int i;
struct pcblock *ps;

/* Initialize the PCB for the main kernel process */
ps = &pcbarr[0];
ps->pcid.node = node;
ps->pcid.id = 0;
ps->pcprio = 30;
ps->extstatus = INUSE;
ps->pciobs = NULL;
ps->parent.node = node >> 2;
ps->parent.node = 0;

/* Put all PCBs in the free queue */
/* after initializing them */
freepcbs = newq ();
for (i=1; i<NBRUSERS+1; i++)  {
        ps = &pcbarr[i];
```

```
        ps->extstatus = 0;
        ps->privsem = newsem (0);
        ps->pciobs = NULL;
        ps->pstk.obj.node = node;
        ps->pstk.obj.id = i;
        ps->pstk.offset = 0;
        put (freepcbs, &ps);
        }

   /* set up the pointers to the user code segments */
ucode[0] = ucode0;  ucode[1] = ucode1;  ucode[2] = ucode2;
}

/*************************************************************************
*       USER - Runs a user process.  The code is reentrant, and
*       each subprocess supervises one active user.
*************************************************************************/
user (i)
int i;  {
struct pcblock *ps;  int id;
ps = &pcbarr[i];

while (TRUE)  {
             /* Wait until there is an active user in the block */
        p (ps->privsem);

          /* Start the process by calling the routine pointed
                to in the PCB program counter  */
        id = (ps->pc).obj.id;
        if (id < 0   ||   id >= NBRUCODES)  error ("Bad pc id.");
        DISABLE
        printf ("*** Process %d %d started on node %d ***\n",
                ps->pcid.node, ps->pcid.id, node);
        ENABLE
        (*ucode[id]) (ps);

          /* When it terminates, deallocate the block */
        DISABLE
        printf ("*** Process %d %d terminated, node %d ***\n",
                ps->pcid.node, ps->pcid.id, node);
        ENABLE
        usrdstry (ps);
        }
}
```

```
/**********************************************************************
*       KERNEL SERVICE REQUEST ROUTINES
*       For purposes of the simulation, each call has an extra
*       parameter at the beginning of the argument list.  This
*       argument is a pointer to the user PCB and is necessary
*       since a subprocess has no space of its own except for
*       the data stack.
**********************************************************************/
#include "defglob"
#include "defkstrucs"

/**********************************************************************
*       PCREATE - Create a child process
**********************************************************************/
pcreate (me, tnode, start, prio, pname)
struct pcblock *me;  NODENBR tnode;  ADDRESS *start;
int prio;  NAME *pname;  {
struct pcblock *ps, *kcrtprc();
struct remcrt *cm;  struct remcrtb *cb;
struct ioblock *iin, *iout, *newiob();
struct hdbfr *h;

if (tnode == node)  {
        /* New process is to be in this node and creation is done
                directly.  No messages are necessary  */
        ps = kcrtusr (start, prio, &me->pcid);
        namecopy (pname, &ps->pcid);
        }
else    {
        /* New process is to be in another node.  Get and fill
                in an IOB to receive the reply message */
        iin = newiob ();
        iin->owner = me;
        iin->ownlink = me->pciobs;
        me->pciobs = iin;
        iin->msglng = sizeof (*cb);
        iin->ioaction = ACTUSER;
        iin->iostat = IOIN | IOWAIT;
        iin->relionbr = -1;

        /* Get and fill in an IOB to send the requesting message */
        iout = newiob();
        h = &iout->msghdr;
        iout->owner = me;
        iout->ownlink = me->pciobs;
        me->pciobs = iout;
        iout->relionbr = 0;

        /* Get a message buffer and fill in the creation request */
        iout->kmptr = newkbf ();
        iout->msgaddr.offset = cm = (iout->kmptr)->kbfr;
        iout->msglng = sizeof (*cm);
        cm->rreq = PCREATE;
        cm->rprio = prio;
        addrcopy (&cm->rstart, start);
        cm->retcio = -1;
```

```
            /* Fill in the message header and put the request message
                    on the output queue */
            iout->ioaction = DISCARD;
            iout->iostat = IOOUT | IOWAIT;
            h->tonode = tnode;
            h->toionbr = 0;
            h->fromnode = node;
            h->toproc.node = tnode;
            h->toproc.id = 0;
            namecopy (&h->fromproc, &me->pcid);
            put (iooutq, &iout);

            /* Wait for the reply message (Message Handler will 'v'
                    this semaphore to wake me up when it comes) */
            p (me->privsem);

            /* Return process id of child to requesting user */
            cb = iin->msgaddr.offset;
            namecopy (pname, &cb->newchid);
            }
return;
}

/*********************************************************************
*         PSTART - Start up a process
*********************************************************************/
pstart (me, pname)
struct pcblock *me;  NAME *pname;  {
struct pcblock *ps, *kertprc();
struct remstr *cm;  struct remstrb *cb;
struct ioblock *iin, *iout, *newiob();
struct hdbfr *h;
int i, a;

   /* Look up process id in the hash table to find out where it is */
if (pname->node < 0 || pname->node >NBRNODES)
            error ("Bad pname in pstart");
i = hfind (pname, &a);
switch (i)  {
      case HPCB:
            /* Process is in this node and can be started
                    directly.  No messages are necessary  */
            ps = a;
            kstrtusr (ps);
            break;
      case HDEL:
            /* Process is not in hash table.  Target the message to the
                    node mentioned in the process id  */
            a = pname->node;
      case HNODE:
            /* Process is in another node.  Get and fill
                    in an IOB to receive the reply message */
            iin = newiob ();
            iin->owner = me;
            iin->ownlink = me->pciobs;
```

```
        me->pciobs = iin;
        iin->msglng = sizeof (*ch);
        iin->ioaction = ACTUSER;
        iin->iostat = IOIN | IOWAIT;
        iin->relionbr = -1;

          /* Get and fill in an IOB to send the requesting message */
        iout = newiob();
        h = &iout->msghdr;
        iout->owner = me;
        iout->ownlink = me->pciobs;
        me->pciobs = iout;
        iout->relionbr = 0;

          /* Get a message buffer and fill in the creation request */
        iout->kmptr = newkbf ();
        iout->msgaddr.offset = cm = (iout->kmptr)->kbfr;
        iout->msglng = sizeof (*cm);
        cm->rreq = PSTART;
        namecopy (&cm->rproc, pname);
        cm->retsig = -1;

          /* Fill in the message header and put the request message
                on the output queue */
        iout->ioaction = DISCARD;
        iout->iostat = IOOUT | IOWAIT;
        h->tonode = a;
        h->toionbr = 0;
        h->fromnode = node;
        h->toproc.node = a;
        h->toproc.id = 0;
        namecopy (&h->fromproc, &me->pcid);
        put (iooutq, &iout);

          /* Wait for the reply message (Message Handler will 'v'
                this semaphore to wake me up when it comes) */
        p (me->privsem);

          /* Resume the user by returning */
        }
return;
}

/**********************************************************************
*          MSEND - Send a message
**********************************************************************/
msend (me, top, tonbr, lng, msg)
struct pcblock *me;  NAME *top;  int tonbr, lng;  char *msg;  {
struct ioblock *ioh, *newiob(), *tioh, *findioh();
struct kmsg *newkbf();
struct hdbfr *h, *th;
int i, a, l;

  /* Look up in hash table to find where a process is located */
i = hfind (top, &a);
switc  (i)  {
```

```
	case HDEL:
		/* If not in hash table, use the node nbr in the process id */
	    a = top->node;
	case HNODE:
		/* If in a different node, get and fill in an IOB for the
			message */
	    iob = newiob();
	    h = &iob->msghdr;
	    iob->owner = me;
	    iob->ownlink = me->pciobs;
	    me->pciobs = iob;
	    iob->relionbr = 0;
	    if (lng>KMSGSIZ)  error ("Usend msg too long");
	    iob->msglng = lng;
		/* Get a message buffer and copy the message into it
			(This is only necessary in the emulator since the
			(data stacks are moved around.  It is not the plan
			(for X-tree itself.)  */
	    iob->kmptr = newkbf ();
	    iob->msgaddr.offset = (iob->kmptr)->kbfr;
	    bytecopy (iob->msgaddr.offset, msg, lng);

		/* Fill in the message header and put the request message
			on the output queue */
	    iob->ioaction = ACTUSER;
	    iob->iostat = IOOUT | IOWAIT;
	    h->tonode = a;
	    h->toionbr = tonbr;
	    h->fromnode = node;
	    namecopy (&h->toproc, top);
	    namecopy (&h->fromproc, &me->pcid);
	    out (iooutq, &iob);

		/* Wait for completion of the message */
	    p (me->privsem);
	    break;
	case HPCB:
		/* Target is in same node.  Obtain IOB and fill in header
			only and use FINDIOB to find the matching input IOB */
	    iob = newiob();
	    h = &iob->msghdr;
	    h->tonode = node;
	    h->toionbr = tonbr;
	    h->fromnode = node;
	    namecopy (&h->toproc, top);
	    namecopy (&h->fromproc, &me->pcid);
	    if ((tiob = findiob(h,IOIN)) == NULL)  error ("No in-node iob match.");

		/* Copy the header, message and message length into the
			receiving block */
	    th = &tiob->msghdr;
	    tiob->iostat = IOIN | IOBUSY;
	    bytecopy (th, h, sizeof(*h));
	    l = (lng < tiob->msglng) ? lng : tiob->msglng;
	    bytecopy (tiob->msgaddr.offset, msg, l);
	    tiob->msglng = l;
```

```
        tiob->iostat = IOOUT | IODONE;

          /* Pass the receiving block to the message handler for
                disposition and resume the user  */
        put (actionq, &tiob);
        break;
    }
return;
}

/***********************************************************************
*       MRECV - Set up to receive a message
***********************************************************************/
mrecv (me, relnbr, lng, msg)
struct pcblock *me;  int relnbr, *lng;  char *msg;  {
struct ioblock *iob, *newiob();

  /* Get and fill in an ioblock to receive the message.
        Leave the msgaddr blank so that a dynamic buffer
        will be allocated when the message is received. */
iob = newiob();
iob->owner = me;
iob->ownlink = me->pciobs;
me->pciobs = iob;
iob->ioaction = ACTUSER;
iob->iostat = IOIN | IOWAIT;
iob->relionbr = relnbr;

  /* Wait for completion of the message */
p (me->privsem);

  /* Copy the message into the user address space.  Again,
        this is necessary in the present simulator, not in
        the real Xtree */
if (iob->msglng < *lng)  *lng = iob->msglng;
bytecopy (msg, iob->msgaddr.offset, *lng);

  /* Release the IOB */
reliob (iob);
return;
}

/***********************************************************************
*       NPARENT - Return the name of the processes parent
***********************************************************************/
nparent (me, pname)
struct pcblock *me;  NAME *pname;  {
namecopy (pname, &me->parent);
return;
}
```

```
#include <stdio.h>
#include "defglob"
#include "defkstrucs"

/*****************************************************************/
/*  HASH TABLE MANAGER -                                         */
/*****************************************************************/
#define NBRPHASH        16
#define NBRPLOC         20
static struct ploc {                /* A hash table entry */
        struct ploc     *plflink;       /* Links for hash list */
        struct ploc     *plblink;
        struct ploc     *pllruf;        /* Links for LRU management */
        struct ploc     *pllruh;        /* of off-node references */
        NAME    plid;               /* Full name of process */
        int     plstat;             /* If plstat=HNODE, plval=nodenbr */
        int     plval;              /*         where process is */
                                    /* If plstat=HPCB, plval=pointer
                                               to PCB */
        }               plocarr[NBRPLOC], *phashtbl[NBRPHASH], *pfree;
                            /* Header of LRU list of HNODE entries */
static struct   {
        int dummy[2];
        struct ploc     *pllruf;
        struct ploc     *pllruh;
        }               plru;

/*****************************************************************
*       INITHSH - Initialize the hash table
*****************************************************************/
inithsh ()   {
int i;
for (i=0; i<NBRPHASH; i++)  phashtbl[i]=NULL;
for (i=0; i<NBRPLOC; i++)  plocarr[i].plflink = &plocarr[i+1];
plocarr[NBRPLOC-1].plflink = NULL;
plru.pllruf = plru.pllruh = &plru;
pfree = plocarr;
return;
}

/*****************************************************************
*       HFIND - Locate a process and return its status as the
*       function value and the value (node nbr or PCB pointer)
*       in argument a
*****************************************************************/
hfind (n,a)  NAME *n; int *a;  {
struct ploc *pl;
pl = phashtbl[hashn(n)];
while (pl != NULL)  {
        if (namecmp(n,&(pl->plid)) == 0)  {
                switch (pl->plstat)  {
                        case HNODE:
                        case HPCB:
                            *a = pl->plval;
                            return (pl->plstat);
                        default:
                            error ("Bad ploc status.");
```

```
				}
			}
		pl = pl->plflink;
		}
	return (HDEL);
	}

	/**********************************************************************
	*	HUPDATE - Update a process's status in the hash table.
	*	The update type code (delete, set node location hint, or
	*	set pcb pointer) are listed in "defkstrucs"
	**********************************************************************/
	hupdate (n,type,where)  NAME *n;  int type, where;  {
	struct ploc *pl, *plb;
	plb = &phashtbl[hashn(n)];  pl = plb->plflink;
	while (pl != NULL)  {
		if (namecmp(n,&pl->plid) == 0)  break;
		plb = pl;  pl = pl->plflink;
		}
	if (type == HDEL)  {
		if (pl == NULL)  return;
		hdelnk(pl);
		return;
		}
	if (pl == NULL)  {
		plb->plflink = pl = getploc();
		pl->plblink = plb;
		namecopy (&pl->plid,n);
		}
	switch (type)  {
	case HNODE:
	case HPCB:  pl->plval = where;  break;
	default:  error ("Invalid type in hupdate.");
	}
	pl->plstat = type;
	htouch (pl);
	return;
	}

	/**********************************************************************
	*	HASHN - the hashing function
	**********************************************************************/
	static hashn (n)  NAME *n;  {
	char *c;  int h, i;
	for (h=0,c=n,i=0; i<sizeof(NAME); i++,c++)
		h = h ^ (*c & 017) ^ ((*c>>4) & 017);
	return (h);
	}

	/**********************************************************************
	*	HDELNK - Delink a hash table entry from both the hash list
	*	and the LRU list
	**********************************************************************/
	static hdelnk (pl)  struct ploc *pl;  {
	if (pl->plflink != NULL)  (pl->plflink) ->plblink = pl->plblink;
	if (pl->plblink != NULL)  (pl->plblink) ->plflink = pl->plflink;
```

```
if (pl->pllruf != NULL)  (pl->pllruf) ->pllrub = pl->pllrub;
if (pl->pllrub != NULL)  (pl->pllrub) ->pllruf = pl->pllruf;
pl->plflink = pl->plblink = pl->pllruf = pl->pllrub = NULL;
pl->plflink = pfree;
pfree = pl;
return;
}

/*********************************************************************
*       HTOUCH - Update an entries location in the LRU list
*********************************************************************/
static htouch (pl)  struct ploc *pl;  {
if (pl->pllruf != NULL)  (pl->pllruf) ->pllrub = pl->pllrub;
if (pl->pllrub != NULL)  (pl->pllrub) ->pllruf = pl->pllruf;
pl->pllrub = pl->pllruf = NULL;
if (pl->plstat == HNODE)  {
        pl->pllruf = plru.pllruf;  plru.pllruf = pl;
        (pl->pllruf)->pllrub = pl;  pl->pllrub = &plru;
        }
return;
}

/*********************************************************************
*       GETPLOC - Get a fresh entry to add to the has table.
*       If no unused ones are available, remove the Least Recently
*       Used HNODE entry.
*********************************************************************/
static getploc ()  {
struct ploc *pl;
while (pfree == NULL)  hdelnk (plru.pllrub);
pl = pfree;
pfree = pl->plflink;
pl->plflink = NULL;
pl->plblink = NULL;
pl->pllruf = NULL;
pl->pllrub = NULL;
nilname (&pl->plid);
pl->plstat = 0;
pl->plval = 0;
return (pl);
}

#ifdef SAIFST
main ()  {
int i, *ptr, arr[4];  char cmnd[10];
inithsh ();
while (TRUE)  {
printf ("command:");
i = scanf ("%s", cmnd);
if (i==EOF)  {putchar ('\n');  exit();}
if (strcmp(cmnd,"displaya") == 0)
        printf ("plocarr: %o\nphashtbl: %o\npfree: %o\nplru: %o\n",
                    plocarr, phashtbl, &pfree, &plru);
else if (strcmp(cmnd,"display") == 0)  {
        while (TRUE)  {
                scanf ("%o", &ptr);
```

```
                        if (ptr==0)  break;
                        for (i=0; i<8; i++)  {printf ("%o ", *ptr); ptr++;}
                        putchar ('\n');
                        }
                }
        else if (strcmp(cmnd,"hfind") == 0)  {
                scanf("%o %o %o", arr, arr+1, arr+2);
                i = hfind(arr, arr+2);
                printf("%o %o %o %o\n", i, arr[0], arr[1], arr[2]);
                }
        else if (strcmp(cmnd,"hupdate") == 0)  {
                scanf("%o %o %o %o", arr, arr+1, arr+2, arr+3);
                hupdate(arr, arr[2], arr[3]);
                printf("%o %o %o %o\n", arr[0], arr[1], arr[2], arr[3]);
                }
        }
}
#endif
```

```
/*********************************************************************
*       Miscellaneous routines called throughout the kernel emulator
*********************************************************************/
#include <stdio.h>
#include "defglob"
#include "defkstrucs"

                /*  copy a byte string  */
bytecopy (a1, a2, l)
char *a1, *a2;  int l;  {
int i;
for (i=0; i<l; i++)
                *(a1++) = *(a2++);
return;
}


                /* Display an area of memory as characters       */
show (c,l)
char *c;  int l;  {
char *i;  int ch;
for (i=c; i<c+l; i++) {
                if ((*i >= ' ') && (*i <= '~'))
                        putchar (*i);
                else  {
                        ch = *i;
                        printf ("\\%o", ch);
                        }
                }
putchar ('\n');
}

                /* Fetch a character from a given global ADDRESS */
char cfetch (a)
ADDRESS *a;       {
char c;
c = *(a->offset);
return (c);
}

                /* Fetch an integer from a given global ADDRESS */
ifetch (a)
ADDRESS *a;       {
int i;
i = *(a->offset);
return (i);
}

                /* Print an error message and abort      */
error(message)
char message[];
{
    printf("\nError in node %d: %s\n", node, message);
    abort();
}
```

```
		/* Set a message header to zeroes */
nilhdr (h)
struct hdbfr *h;  {
h->tonode = 0;
nilname (&(h->toproc));
h->tojonbr = 0;
h->fromnode = 0;
nilname (&(h->fromproc));
return;
}

		/* Copy an ADDRESS from a2 to a1 */
addrcopy (a1,a2)
ADDRESS *a1, *a2;  {
namecopy (&(a1->obj), &(a2->obj));
a1->offset = a2->offset;
return;
}

		/* Set an ADDRESS to zeroes */
niladdr (a)
ADDRESS *a;  {
nilname (&(a->obj));
a->offset = 0;
return;
}

		/* Check to see if an ADDRESS is zero */
addrnil (a)
ADDRESS *a;        {
if ((namenil(&(a->obj))) && (a->offset==0))
		return (TRUE);
else
		return (FALSE);
}

		/* Increment an ADDRESS by the length of an integer */
incaddr (a)
ADDRESS *a;        {
a->offset += 2;
return;
}

		/* Copy an global NAME from n2 to n1 */
namecopy (n1,n2)
NAME *n1, *n2;  {
n1->node = n2->node;
n1->id = n2->id;
return;
}

		/* Zero out a global NAME       */
nilname (name)
NAME *name;  {
name->node = 0;
name->id = 0;
```

```
}

        /* Check to see if an global NAME is zero */
namenil (n)
NAME *n;        {
if ((n->node==0) && (n->id==0))
        return (TRUE);
else
        return (FALSE);
}

        /* Compare two global NAMES */
namecmp (n1,n2)
NAME *n1, *n2;   {
if (n1->node < n2->node)  return (-1);
if (n1->node > n2->node)  return (1);
if (n1->id < n2->id)  return (-1);
if (n1->id > n2->id)  return (1);
return (0);
}
```

```
/*******************************************************************
*       Predefined user code segments
*******************************************************************/
#include <stdio.h>
#include "defglob"
#include "defint"

/*******************************************************************
*       Code segment 0 -
*               Create and start a process on node 1 using
*                       code segment 1
*               Wait for a message
*               When it comes, print it and quit
*******************************************************************/
ucode0 (ps)     struct pcblock *ps;     {
ADDRESS addr1;  NAME pname1;
char msg[25];  int lgth;
DISABLE
printf ("Ucode 0 executed on node %d.\n", node);
ENABLE
addr1.obj.node = 1;
addr1.obj.id = 1;
addr1.offset = 0;
pcreate (ps, 1, &addr1, 2, &pname1);
pstart (ps, &pname1);
lgth = 25;
mrecv (ps, 3, &lgth, msg);
DISABLE
printf ("Msg received by BIG was:\n");
ENABLE
show (msg, lgth);
return;
}

/*******************************************************************
*       Code segment 1 -
*               Create and start a process on node 2
*                       using code segment 2
*               Send a msg to its parent (msg = "It works!")
*               Exit
*******************************************************************/
ucode1 (ps)     struct pcblock *ps;     {
ADDRESS addr1;  NAME pname1, pnamef;
char msg[25];  int lgth;
DISABLE
printf ("Ucode 1 executed on node %d.\n", node);
ENABLE
addr1.obj.node = 1;
addr1.obj.id = 2;
addr1.offset = 0;
pcreate (ps, 2, &addr1, 2, &pname1);
pstart (ps, &pname1);
pparent (ps, &pnamef);
DISABLE
printf ("Ucode 1 about to send msg.\n");
```

```
ENABLE
msend (ps, &nnamef, 3, 9, "It works:");
return;
}

/**********************************************************************
*        Code segment 2
*                Prints out that it executed
*                Then quits
**********************************************************************/
ucode2 (ps)      struct pcblock *ps;      {
DISABLE
printf ("Ucode 2 executed on node %d.\n", node);
ENABLE
}
```

```
/**********************************************************************
*        FIFO Queue Routines
**********************************************************************/
#include "defqsem"

struct  qheadr  qs[QCNT];
int     qctr;
char    *nil    = 0177777;

/**********************************************************************
*        QHEADR - allocate and initialize new queue header
**********************************************************************/
struct  qheadr  *newq() {
struct qheadr *q;
if (qctr >= QCNT)
        error ("No more queues in newq.");
q = &qs[qctr];
qctr++;
q->count = newsem(0);
q->mutex = newsem(1);
q->head = nil;
q->tail = nil;
return (q);
}

/**********************************************************************
*        GET - Pop an element from the top of the queue
**********************************************************************/
struct  qform   *get(q)
struct qheadr *q;        {
struct qform *obj;
if (q==0 || q==nil)
        error ("Bad argument to get.");
p(q->count);
p(q->mutex);
if (q->head == nil)
        error ("Bad queue count in get.");
obj = q->head;
q->head = (q->head)->qlink;
if (obj == q->tail)
        q->tail = nil;
v(q->mutex);
obj->qlink = nil;
return (obj);
}

/**********************************************************************
*        PUT - Add an element to the bottom of the queue
**********************************************************************/
put (q,obj)
struct  qheadr  *q;
struct  qform   **obj;  {
if (q==nil || *obj==nil || q==0 || *obj==0)
        error ("Bad argument to put.");
p(q->mutex);
if (q->head == nil)
```

```
		q->head = *obj;
	else
		(q->tail)->qlink = *obj;
	q->tail = *obj;
	(*obj)->qlink = nil;
	*obj = nil;
	v(q->mutex);
	v(q->count);
	}
```

```
/************************************************************************
*       Semaphores and Subprocess Definition
************************************************************************/
#include "defasem"
#include "defint"

char *spnil = 0177777;

struct sem sems[SEMMAX];
int semctr 0;

struct sp sps[SPMAX];
int spctr = 1;
struct sp *spcur = &sps[0];
struct sp *spra = 0177777;

/************************************************************************
*       NEWSEM - Allocate a semaphore withe initial value n
************************************************************************/
struct sem *newsem(n)
int n;
{
    extern struct sem sems[];
    extern int semctr;
    struct sem *s;

    if ( semctr >= SEMMAX )
        error("newsem: no more semaphores to allocate");
    s = &sems[semctr];
    semctr = semctr + 1;
    s->qhead = spnil;
    s->value = n;
    return(s);
}

/************************************************************************
*       P - The "P" semaphore operation with queue based waiting
************************************************************************/
p(s)
struct sem *s;
{
    extern struct sp *spget();

    if ( !semchk(s) )
        error("p: invalid semaphore address");
    DISABLE
    s->value = s->value - 1;
    if ( s->value < 0 )
    {
        spsave(&(spcur->state));
        spput(&(s->qhead), spcur);
        spcur = spget(&spra);
        if ( spcur == spnil )
            error("p: no subprocesses available to run");
        spstart(&(spcur->state));
    }
```

```
	ENABLE
}

/**********************************************************************
*	V - The 'V' semaphore operation
**********************************************************************/
v(s)
struct sem *s;
{
	extern struct sp *spget();
	struct sp *new;

	if ( !semchk(s) )
		error("v: invalid semaphore address");
	DISABLE
	s->value = s->value + 1;
	if ( s->value <= 0 )
	{
		new = spget(&s->qhead);
		if ( new == spnil )
			error("v: unexpected empty semaphore queue");
		spprio(new);
	}
	ENABLE
}

/**********************************************************************
*	SEMCHK - Check a semaphore pointer for validity
**********************************************************************/
boolean semchk(s)
struct sem *s;
{
	extern struct sem sems[];
	extern int semctr;
	int t, q;

	t = s;
	q = sems;
	return ( s >= sems &&
		s < sems + semctr &&
		((t - q) % sizeof sems[0]) == 0 );
}

/**********************************************************************
*	SPPRIO - start the next subprocess if 'new' priority is
*		higher than the current priority
**********************************************************************/
spprio(new)
struct sp *new;
{
	if ( new->priority >= spcur->priority )
	{
		spsave(&(spcur->state));
		spput(&sprq, spcur);
		spcur = new;
		spstart(&(spcur->state));
```

```
    }
    else
    {
        spput(&sprq, new);
    }
}

/*********************************************************************
*       SPFORK - create a new subprocess of given priority
*********************************************************************/
boolean spfork(prio)
{
    extern struct sp sps[];
    extern int spctr;
    struct sp *new;

    if ( spctr >= SPMAX )
        error("spfork: no more subprocess slots");
    new = &sps[spctr];
    spctr = spctr + 1;
    new->priority = prio;
    spsave(&(new->state));
    spprio(new);
    return(true);
}

/*********************************************************************
*       SPEXIT - permanently terminate the current subprocess
*********************************************************************/
spexit()
{
    spcur = spget(&sprq);
    if ( spcur == spnil )
        error("spexit: no subprocesses ready to run");
    spstart(&(spcur->state));
}


/*********************************************************************
*       SPGET - Internal routine to get the next subprocess from
*       a semaphore queue
*********************************************************************/
struct sp *spget(q)
struct sp **q;
{
    struct sp *subproc;

    subproc = *q;
    if ( subproc == spnil )
        return(subproc);
    *q = subproc->splink;
    subproc->splink = spnil;
    return(subproc);
}

/*********************************************************************
```

```
*       SPPUT - Internal routine to put a subprocess onto a queue,
*       in priority order
***********************************************************************/
spput(q, proc)
struct sp **q;
struct sp *proc;
{
    struct sp *t;
    if ( (*q == spnil) ||
         (proc->priority >= (*q)->priority) )
    {
        proc->splink = *q;
        *q = proc;
    }
    else
    {
        t = *q;
        while ( (t->splink != spnil) &&
                (proc->priority <= (t->splink)->priority) )
        {
            t = t->splink;
        }
        proc->splink = t->splink;
        t->splink = proc;
    }
}
```

```c
/*********************************************************************
*       Subprocess swapping routines, internal to subprocess
*       management
*********************************************************************/
#include "defqsem"


/*********************************************************************
*       SPSAVE - Store away the current subprocess's data stack
*********************************************************************/
spsave(state)
struct stateform *state;
{
    int i;
    int *t;
    int size;

    t = &i + 4;
    t = *t;
    t = t - 4;
    size = t;
    size = -size / 2;
    if ( size > STACKMAX )
        error("spsave: caller's stack too large to save");
    state->stkptr = t;


    for ( i = 0 ; i < size ; i = i + 1 )
        state->stack[i] = t[i];
}

/*********************************************************************
*       SPSTART - Restore a subprocess's data stack from save
*       storage
*********************************************************************/
spstart(state)
struct stateform *state;
{
    int i;
    int *t;
    int size;

    t = state->stkptr;
    size = t;
    size = -size / 2;
    if ( size > STACKMAX || size == 0 )
        error("spstart: argument is not a valid stateform");
    if ( t <= &i + 8 )
        spstart(state);


    for ( i = 0 ; i < size ; i = i + 1 )
        t[i] = state->stack[i];
```

```
    spcaller(t);
    return(false);
}

/***********************************************************************
*        SPCALLER - Tricky routine to force return to restored
*        stack frame
***********************************************************************/
spcaller(t)
int *t;
{
    int i;
    (&i)[4] = t + 4;
}
```

NAME
    fork - spawn new process

SYNOPSIS
    fork( )

DESCRIPTION
    Fork is the only way new processes are created. The new process's core image is a copy of that of the caller of fork. The only distinction is the fact that the value returned in the old (parent) process contains the process ID of the new (child) process, while the value returned in the child is 0. Process ID's range from 1 to 30,000. This process ID is used by wait(2).

    Files open before the fork are shared, and have a common read-write pointer. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

SEE ALSO
    wait(2), exec(2)

DIAGNOSTICS
    Returns -1 and fails to create a process if: there is inadequate swap space, the user is not super-user and has too many processes, or the system's process table is full. Only the super-user can take the last process-table slot.

ASSEMBLER (PDP-11)
    (fork = 2.)
    sys fork
    (new process return)
    (old process return, new process ID in r0)

    The return locations in the old and new process differ by one word. The C-bit is set in the old process if a new process could not be created.

NAME
kill - send signal to a process

SYNOPSIS
```
kill(pid, sig);
```

DESCRIPTION
Kill sends the signal sig to the process specified by the process number in r0. See signal(2) for a list of signals.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all other processes in the sender's process group; see tty(4).

If the process number is -1, and the user is the super-user, the signal is broadcast universally except to processes 0 and 1, the scheduler and initialization processes, see init(8).

Proccesses may send signals to themselves.

SEE ALSO
signal(2), kill(1)

DIAGNOSTICS
Zero is returned if the process is killed; -1 is returned if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist.

ASSEMBLER (PDP-11)
```
(kill = 37.)
(process number in r0)
sys kill; sig
```

NAME
pipe - create an interprocess channel

SYNOPSIS
```
pipe(fildes)
int fildes[2];
```

DESCRIPTION
The pipe system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor fildes[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor fildes[0] will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent **fork** calls) will pass data through the pipe with **read** and **write** calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

SEE ALSO
sh(1), read(2), write(2), fork(2)

DIAGNOSTICS
The function value zero is returned if the pipe was created; -1 if too many files are already open. A signal is generated if a write on a pipe with only one end is attempted.

BUGS
Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.

ASSEMBLER (PDP-11)
```
(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)
```

NAME

signal - catch or ignore signals

SYNOPSIS

```
#include <signal.h>

(*signal(sig, func))();
(*func)();
```

DESCRIPTION

A signal is generated by some abnormal event, initiated either by user at a terminal (quit, interrupt), by a program error (bus error, etc.), or by request of another program (kill). Normally all signals cause termination of the receiving process, but a signal call allows them either to be ignored or to cause an interrupt to a specified location. Here is the list of signals with names as in the include file.

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction (not reset when caught)
SIGTRAP	5*	trace trap (not reset when caught)
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
	16	unassigned

The starred signals in the list above cause a core image if not caught or ignored.

If func is SIG_DFL, the default action for signal sig is reinstated; this default is termination, sometimes with a core image. If func is SIG_IGN the signal is ignored. Otherwise when the signal occurs func will be called with the signal number as argument. A return from the function will continue the process at the point it was interrupted. Except as indicated, a signal is reset to SIG_DFL after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another signal call.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a read or write(2) on a slow device (like a terminal;

but not a file); and during pause or wait(2). When such a signal occurs, the saved user status is arranged in such a way that when return from the signal-catching takes place, it will appear that the system call returned an error status. The user's program may then, if it wishes, re-execute the call.

The value of signal is the previous (or initial) value of func for the particular signal.

After a fork(2) the child inherits all signals. Exec(2) resets all caught signals to default action.

SEE ALSO

kill(1), kill(2), ptrace(2), setjmp(3)

DIAGNOSTICS

The value (int)-1 is returned if the given signal is out of range.

BUGS

If a repeated signal arrives before the last one can be reset, there is no chance to catch it.

The type specification of the routine and its func argument are problematical.

[illegible] VAX-11, odd values for func are the same as SIG_IGN.

ASSEMBLER (PDP-11)

(signal = 48.)
sys signal; sig; label
(old label in r0)

If label is 0, default action is reinstated. If label is odd, the signal is ignored. Any other even label specifies an address in the process where an interrupt is simulated. An RTI or RTT instruction will return from the interrupt.

NOTES (VAX-11)

The following defines the mapping of hardware traps to signals:

Arithemetic traps:

Integer overflow	SIGFPE
Integer division by zero	SIGFPE
Floating overflow	SIGFPE
Floating underflow	SIGFPE
Floating division by zero	SIGFPE
Decimal division by zero	SIGFPE
Decimal overflow	SIGFPE
Subscript-range	SIGFPE

Access control (i.e. protection violation)	
except length violation	SIGBUS
Translation not valid, and	
Length access control	SIGSEGV
Reserved instruction	SIGILL
Customer-reserved instr.	SIGEMT
Reserved operand	SIGILL
Reserved addressing	SIGILL
Trace pending	SIGTRAP
Bpt instruction	SIGTRAP
Compatibility-mode	SIGEMT
Chme	SIGSEGV
Chms	SIGSEGV
Chmu	SIGBUS

APPENDIX D - Instructions for Running the Emulator

The emulator code, in both source and executable forms, is in the directory '/b/xtree/neil/kersim'. The instructions in this appendix assume that this is the current working directory.

1. Running the Emulator

The terminal operator must start the emulator by entering the command

```
% xtree
```

The system will start up the kernel in each node (UNIX process) and prompt when its ready for commands:

```
command:
```

To start the execution of the first user process, two commands must be entered from the terminal:

1. To create a process to run the code, enter

```
create node prio pgm
  where node = node nbr on which to create the process
        prio = the new process's priority
        pgm = the user code nbr of the desired program
```

The system will print a process name to be used in the start command.

2. To start the execution of the process, enter

```
start name
  where name = the 2-part process name displayed when
               the process was created.
```

For example, the simple program currently in the emulator (section 5.1) can be started by executing user code 0 on node 3 with priority 5:

```
command:create 3 5 0
*** Process created on node 3, named 3 1 ***
command:start 3 1
```

2. Coding and Linking Your Own User Programs

User programs are written in C and are run as subroutines of a kernel process (see section 3.3). User programs

are restricted in the following ways.

1. No input operations are allowed. Any data needed must be received via messages.

2. All variables and data must be local. No externals are allowed.

3. Printing statements may be included, but should be framed by the macros

```
DISABLE
printf (...);
ENABLE
```

   to prevent a process switch when the stack is too large for the data stack save areas (section 4.2).

4. Each user program receives an integer argument. This argument must be used as the first argument to all kernel calls from the user program.

The kernel calls discussed in section 3.3 may be used. Specific examples in the C language can be found in appendix B, file 'sAucodes.c'.

UNIX does not support dynamic subroutine linkage. Consequently, any user programs must be statically linked with the emulator before the emulator is executed. Within the emulator, each user program is referred to by a 'user code number' (see the third argument of the 'create' command).

The inclusion of new user programs takes three steps:

1. Put the source for the user programs into the file 'sAucodes.c'.

2. Update the user initialization code (file 's6pman.c', routine 'initusr') to assign the program names to the table of routine pointers (array 'ucode[]'). The index to which a program name is assigned is the user code number.

3. Execute the UNIX command 'make'. This will do all the necessary compilation and linking automatically.

APPENDIX E - Material on Future Work

1. X-TREE PROCESS MODEL

In the full X-tree system with paging, how is a user process in the X-tree to be represented and manipulated? In essence, a process is represented by two blocks, the Process Control Block (PCB) and the Process Work Object (PWO).

The PCB is relatively small and is maintained in the local memory of the node in which the process is running. It contains the information needed by the kernel to control the process. In particular, the Program Counter (PC) and Data Stack Pointer (DSP) are kept here.

The PWO resides in the global memory (or the backing store if you prefer to call it that) and is paged into the node as required. It contains all of the working data storage for the process and is consolidated into a single object which can be viewed as having two parts. The low address part of the PWO is fixed in size and holds the process data which is not needed in the PCB. For example, the register contents for the process would be kept here.

The second part of the PWO is the data stack for the process. It extends from the top of part 1 in an open ended fashion. This stack is used in the conventional manner for data storage and for block entry and exit.

```
 PCB (in node)
  ----------
 :          :
 :   PC     :
 :          :
 :   DSP ---:---------
 :          :         :
 :   ...    :         :        PWO (on disk)
  ----------          :       :              :
                      :       :              :
                       ------>: top of stack :
                              :              :
                              :--------------:
                              : registers,   :
                              :   etc.       :
                              :              :
                               --------------
```

Note especially that there is no notion of a block of code which 'belongs' to the process. There are two reasons

for this:

1. All X-tree code is pure code and is not modifiable. So there is no need for a separate copy for each user.

2. Linking of object modules will be done dynamically at run time as code modules are called.

As a result, all code is located and paged in to the node dynamically as needed and there is no requirement to have separate user copies of any code.

Concomitant with the lack of a separate 'code image' for a process is the fact that the PC and DSP are not just simple local addresses, but are full global addresses complete with global object names and offsets. As a result, the PC identifies the object which contains the current code for the process. Similarly, the DSP serves the dual function of stack pointer and pointer to the PWO (this is the DSP with the offset masked to zero).

Since the PWO for a process never changes, the 'object name' portion of the DSP is constant for the life of the process. The PC, on the other hand, may change with some frequency as code in new objects is invoked. In this case, the old PC (including object name) must be stored in the data stack to be restored upon return.

Conceptually then, the page table in the node may be viewed as a table of translations from global addresses to local addresses:

OBJECT NAME		OFFSET	LOCAL PAGE #
Node #	Id #	Page #	

Of course, this table is implemented using hashing in order to reduce the search time. Additionally a translation look-aside buffer is used to eliminate the search for most accesses in a cache-like manner.

If the node numbers in the system were fixed length, then this would suffice. However, the node numbers in the final X-tree may be unbounded in length so a different scheme is necessary and is outlined below.

In this new scheme, all the active object names for a process are centralized into one place (currently called the C-list for lack of a better name). This could be included

as part of the PWO or as an extension to the PCB. The varying length node numbers could then be managed as a heap (either one per process or as a central heap for each node). The key here is that each object may now be be referred to by a fixed length index into the C-list and the process can be uniquely identified by it's index into the table of PCBs. Taken together the PCB and C-list indices are a fixed length field which uniquely identifies an object. The TLB then looks like:

```
 --------------------------------------------------------------
 |                                          | Local Page #     |
 |------------------------------------------|                  |
 | Proc ID    | C-list ind    | Page #      |                  |
 |------------|---------------|-------------|------------------|
 |            |               |             |                  |
 |            |               |             |                  |
```

Of course, if the page reference is not found in the TLB, the software must first follow the chains outlined above to determine the object requested, then look up the local page number in the same manner as before. The local address is then given to the TLB.

2. A MODULA Clock/Delay Driver

```
(************************************************************
    CLOCKDRIVER device module: handles clock interrupts
    and defines the means for delaying a user process
    for a specific length of time.
************************************************************)


device module CLOCKDRIVER [CLOCK-PRIORITY];
        define  delay;
        use     NBRPROCS, awaken,
                pcbptr, pcbtype, pcb;
        const   WQSIZE = NBRPROCS+1;
        (****************************************************
            Defines a queue of processes (managed in a
            circular fashion) sorted into the order in
            which they are to be activated.  Each entry
            contains a PCB pointer and the proper time to
            delay it after the previous process has been
            activated.
        ****************************************************)
        var     q: array 0:WQSIZE-1 of record
                        nteger;
                end;
                f, b: integer;  {front and back pointers}
                ct: integer;    {length of queue}
```

```
procedure delay (proc:pcbptr; n:integer);
        var p, r, t: integer;
        begin
        if n>0 then
                p := f;  r := b;  t := 0;
                while (p<>b) and (t+q[p].dt<n) do
                        inc(t,q[p].dt);
                        p := (p+1) mod WQSIZE;
                end;
                while r<>p do
                        q[r] := q[(r-1) mod WQSIZE];
                        r := (r-1) mod WQSIZE;
                end;
                q[p].dt := n-t;
                q[p].pcb := proc;
                pcb[proc].psw := pcb[proc].psw or [1,3];
                dec(q[(p+1) mod WQSIZE].dt, q[p].dt);
                b := (b+1) mod WQSIZE;
                inc (ct);
        else
                awaken (proc);
        end;
        end delay;


process clock [CLOCK-INTERRUPT-ADDRESS];
        var     csr [CLOCK-STATUS-REGISTER-ADDRESS]: bits;
        begin
        loop
                csr [6] := true;
                doio;
                loop
                        when ct=0 exit;
                        dec(q[f].dt);
                        when q[f].dt>0 exit;
                        awaken (q[f].pcb);
                        f := (f+1) mod WQSIZE;
                        dec (ct);
                end;
        end;
        end clock;


begin
        f := 0;
        b := 0;
        ct := 0;
        clock;
end CLOCKDRIVER;
```